5-2006

Trees with Unique Minimum Locating-Dominating Sets.

Stephen M. Lane
East Tennessee State University

Trees with Unique Minimum Locating-Dominating Sets

A thesis
presented to
the faculty of the Department of Mathematics
East Tennessee State University

In partial fulfillment of
the requirements for the degree
Master of Science in Mathematical Sciences

---

by
Stephen M. Lane
May 2006

Teresa Haynes, Ph.D., Chair
Robert Gardner, Ph.D.
Debra Knisley, Ph.D.

Keywords: locating-domination, locating-dominating set

ABSTRACT

Trees with Unique Minimum Locating-Dominating Sets

by

Stephen M. Lane

A set S of vertices in a graph $G = (V, E)$ is a locating-dominating set if S is a dominating set of G, and every pair of distinct vertices $\{u, v\}$ in $V - S$ is located with respect to S, that is, if the set of neighbors of u that are in S is not equal to the set of neighbors of v that are in S. We give a construction of trees that have unique minimum locating-dominating sets.

DEDICATION

I dedicate this thesis to Hüsker Dü and *Die Siedler von Catan*.

ACKNOWLEDGEMENTS

I thank my Mom and Dad, and Drs Teresa Haynes, Robert Gardner, Debra Knisley, George Poole, and Frederick Norwood for helping make my experience at ETSU so enjoyable and rewarding. I also thank the Math Lab tutorees and the students of my classes for doing the same.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION AND MOTIVATION

Consider a building with rooms that are to be monitored. We have expensive monitoring devices that we can place in or adjacent to rooms that we wish to monitor. Additionally, we want a monitored room to be uniquely specified by a set of monitoring devices; that is, if we receive signals from devices (say) B, M, and P, we want to know that room (say) 231 is the room monitored by that set of devices. Since the monitoring devices are expensive, we also want to minimize the number of devices used.

This situation can be modeled as a graph, with vertices of the graph representing building rooms, and edges placed between adjacent rooms. If a monitoring device can monitor the room it is in and all adjacent rooms, the problem of minimizing the number of monitoring devices becomes the graph theory problem of *locating-domination*, which is the subject of this thesis.

Let $G = (V, E)$ be a graph. For a vertex v in V, the *open neighborhood of* v, denoted $N(v)$, is the set of vertices in V adjacent to v. (Our definitions follow [5] and [19].) Let S be a subset of vertices of V. Then the *open neighborhood of* v *with respect to* S, denoted $N_S(v)$, is the set of vertices in S that are adjacent to v. Note that $N_S(v) = N(v) \cap S$.

The subset S is called a *dominating set* of a graph $G = (V, E)$ if every vertex in $V - S$ has a neighbor in S, or equivalently, if $N_S(v)$ is nonempty for every vertex v in $V - S$. For a graph G, the cardinality of a smallest dominating set of G is called the *domination number* of G, and is denoted by $\gamma(G)$, or simply by γ, if the graph G is understood. Any dominating set of G of cardinality $\gamma(G)$ is called a $\gamma(G)$-set.

Let $G = (V, E)$ be a graph, L be a dominating set of G, and L have the additional property that for every (distinct) pair of vertices v, w in $V - L$, $N_L(v) \neq N_L(w)$. Then L is called a *locating-dominating set*, or LD-set of G, or an LD(G)-set.

As with the domination number $\gamma(G)$, the *locating-domination number* $\gamma_L(G)$ is the minimum cardinality among all LD-sets of G. Then any LD(G)-set of cardinality $\gamma_L(G)$ is called a $\gamma_L(G)$-set.

We illustrate locating-domination by finding $\gamma_L(P_6)$ (see Figure 1). Let $P_6 = (a, b, c, d, e, f)$, and let L be our proposed $\gamma_L(P_6)$-set. At least one of $\{a, b\}$ is in L, or otherwise a would not be dominated, and so L would not be a $\gamma_L(P_6)$-set. Choose a to be in L. As we want to minimize the cardinality of L, choose b and c to not be in L. Then d is in L, or else c would not be dominated. Finally, at least one of $\{e, f\}$ is in L for f to be dominated. Choose e to be in L. Then $L = \{a, d, e\}$.

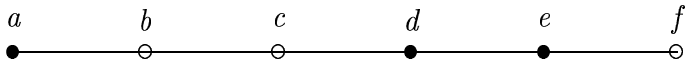


Figure 1: $\gamma_L(P_6) = 3$. The shaded vertices form a minimum locating-dominating set.

We next verify that L is locating: $N_L(b) = \{a\}$, $N_L(c) = \{d\}$, and $N_L(f) = \{e\}$, so L is indeed locating, and L is an LD(P_6)-set of cardinality 3. Thus $\gamma_L(P_6) \leq 3$.

Next, note that there can be no $\gamma_L(P_6)$-set of cardinality 2. For $\gamma(P_6) = 2$, and the only $\gamma(P_6)$-set is $S = \{b, e\}$. But $N_S(a) = \{b\} = N_S(c)$, and $N_S(d) = \{e\} = N_S(f)$, so S is not locating. Hence, $\gamma_L(P_6) > 2$, and $\gamma_L(P_6) = 3$.

The graph P_6 has γ_L-sets other than $\{a, d, e\}$. For example, by symmetry, $\{b, c, f\}$ is also a $\gamma_L(P_6)$-set.

On the other hand, P_5 has exactly one γ_L-set. Let $P_5 = (a, b, c, d, e)$; then $\{b, d\}$ is the *unique* γ_L-set of P_5.

There have been considerable successes in characterizing families of graphs that

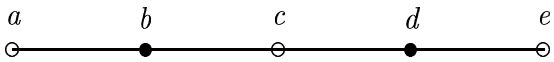


Figure 2: $\gamma_L(P_5) = 2$. P_5 is a ULD-tree, with $\{b, d\}$ as its only γ_L-set.

have unique γ_X-sets, where γ_X may represent domination, total domination, double domination, paired domination, etc. [18, 21]. This prior research serves as motivation for our research.

A *tree* is a connected graph with no cycles. Our goal is to characterize trees that have unique minimum locating-dominating sets. We denote such trees as *ULD-trees*. As shown above, there is at least one such non-trivial ULD-tree: P_5.

Some definitions: a *path* in a graph (V, E) is an ordered list of vertices $(v_1, v_2, \ldots, v_k) \subset V$ such that for every $1 \leq i \leq k-1$, vertices v_i and v_{i+1} are adjacent (equivalently, $(v_i, v_{i+1}) \in E$). A *component* C of a graph $G = (V, E)$ is a maximal set of vertices of V such that there exists a path between any two vertices of C.

If there exists a path between two vertices u, v in a graph, we say that u and v are *connected*. If every pair of vertices in a graph are connected, we say that the graph itself is *connected*. This is equivalent to saying that the graph consists of one component. Otherwise, the graph consists of multiple *disconnected* components.

A graph G is *complete* if every pair of vertices in G are adjacent. For every positive integer n, there is one complete graph on n vertices, denoted K_n.

The *complement* of a graph $G = (V, E)$, denoted $\overline{G}$, consists of the set of vertices V, with vertices u and v adjacent if and only if u and v are not adjacent in G. The complement of a complete graph on n vertices K_n, denoted $\overline{K_n}$, consists of a set of n vertices with no edges. The graph $\overline{K_n}$ is also called the *empty graph* on n vertices.

This section summarizes known results on locating-domination.

Note that γ_L is defined for every graph. For a graph $G = (V, E)$, V is an LD(G)-set, as $V - V = \emptyset$, and the conditions for V to be an LD(G)-set are (vacuously) true.

The first description of locating-domination was given by Slater [28]. His initial results, including the exact locating-domination number for various families of graphs, are given below:

Theorem 3.1 *[28] For any graph G, $\gamma_L(G) \geq \gamma(G)$.*

If a graph G has connected components $G_1, G_2, \ldots, G_k$, then $\gamma_L(G) = \gamma_L(G_1) + \gamma_L(G_2) + \cdots + \gamma_L(G_k)$.

For a graph G with n vertices, $\gamma_L(G) = n$ if and only if $G = \overline{K_n}$.

For a graph G with n vertices, $\gamma_L(G) = n - 1$ if and only if G has exactly one nontrivial component, and this component is complete or a star. (See Figure 3.)

A graph G has $\gamma_L(G) = 1$ if and only if $G = K_1$ or K_2.

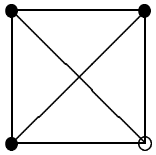


Figure 3: Although $\gamma(K_n) = 1$, $\gamma_L(K_n) = n - 1$.

The *path* on n vertices, denoted P_n, is the graph with vertex set $\{v_1, v_2, \ldots, v_n\}$, where the *end-vertices* v_1 and v_n are adjacent to v_2 and v_{n-1}, respectively, and each other vertex v_i is adjacent to vertices v_{i-1} and v_{i+1}.

The *cycle* on n vertices, denoted C_n, is the path P_n with an additional edge between vertices v_1 and v_n. See Figures 1 and 4.

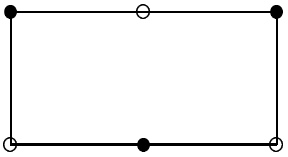


Figure 4: $\gamma_L(C_6) = 3$.

The next theorem gives the exact value of γ_L for paths and cycles.

Theorem 3.2 *[28] Let P_n denote the path with n vertices, and C_n the cycle with n vertices. Then $\gamma_L(P_n) = \gamma_L(C_n) = \lceil 2n/5 \rceil$.*

Before the next theorem, we give some definitions. A *bipartite* graph is one whose vertices can be partitioned into two sets V_1 and V_2, such that there are no edges between vertices in V_1, and no edges between vertices in V_2. The *complete bipartite graph* $K_{s,t}$ is the bipartite graph with s vertices in V_1, t vertices in V_2, and with an edge between every vertex in V_1 and every vertex in V_2.

The *join* of two graphs G and H, denoted $G + H$, is formed by taking the graphs G and H, and then adding an edge between every vertex of G and every vertex of H. With this definition, the bipartite graph $K_{s,t} = \overline{K_s} + \overline{K_t}$.

The *complete k-partite graph* $K_{p_1,p_2,\ldots,p_k}$ is the graph with mutually disjoint vertex sets $V_1, V_2, \ldots, V_k$, with a vertex set V_i containing p_i vertices, and with every vertex in V_i adjacent to every vertex not in V_i.

The next theorem gives the exact value of γ_L for complete bipartite graphs $K_{s,t}$ and complete k-partite graphs.

Theorem 3.3 *[28] If $2 \leq s \leq t$, then $\gamma_L(K_{s,t}) = s + t - 2$. Furthermore, If $p = p_1 + p_2 + \cdots + p_k$ with $2 \leq p_1 \leq p_2 \leq \cdots \leq p_k$, then $\gamma_L(K_{p_1,p_2,\ldots,p_k}) = p - k$.*

A graph can have any number of vertices for a given domination number γ. For example, the star $K_{1,s}$ has domination number 1, and $1 + s$ vertices, for any non-negative integer s. This is not true for the locating-domination number; there is an upper bound on the number of vertices of a graph with a given locating-domination number γ_L:

Theorem 3.4 *[28] Let G be a graph with n vertices. Let $h = \gamma_L(G)$. Then $h+2^h-1 \geq n$. Furthermore, the number of graphs G with $n = h + 2^h - 1$ equals the number of graphs on h vertices.*

This theorem can be used to find the graphs G with $\gamma_L(G) = 1$, as in Theorem 3.1. Consider graphs with $\gamma_L(G) = 2$. By this theorem, the maximum number of vertices in such graphs is $n = 5$, and there are 2 such graphs: P_5 and C_5.

A Nordhaus-Gaddum type result is one which associates the sum or product of a parameter of a graph G with that of its complement $\overline{G}$. The following is a Nordhaus-Gaddum type result on the locating-domination number:

Theorem 3.5 *[28] If G is a graph with $n \geq 2$ vertices, then $\gamma_L(G) + \gamma_L(\overline{G}) \leq 2n - 1$ and $\gamma_L(G)\gamma_L(\overline{G}) \leq n(n-1)$. Furthermore, these bounds are sharp.*

Our research considers trees. The next result is a lower bound for the locating-domination number of trees. Recall that a tree is a connected graph with no cycles.

Theorem 3.6 *[26] For any tree T with n vertices, $\gamma_L(T) > n/3$. Furthermore, there is a tree T with n vertices with $\gamma_L(T) = \lfloor (n+3)/3) \rfloor$.*

A graph G is *r-regular* if every vertex in G is adjacent to r other vertices in G. The following is a lower bound for the locating-domination number for regular graphs:

Theorem 3.7 *[27] If a graph G with n vertices is r-regular, then $\gamma_L(G) \geq 2n/(r+3)$, and this bound is not sharp.*

A variety of lower bounds on a graph's locating-domination number follow:

Theorem 3.8 *[28] If G is a graph with n vertices, $h = \gamma_L(G)$, and Δ is the maximum degree of a vertex in G, then $n \leq h + \sum_{i=1}^{\Delta} \binom{h}{i}$.*

Theorem 3.9 *[25] Let G be a graph with n vertices that does not contain a subdivided K_5 or $K_{2,3}$ as a subgraph. Then $n \leq (7\gamma_L(G) - 3)/2$. (Or $\gamma_L(G) \geq (2n+3)/7$.)*

Theorem 3.10 *[25] Let G be a graph with n vertices that does not contain a subdivided K_4 or $K_{3,3}$ as a subgraph. Then $n \leq 4\gamma_L(G) - 3$ for $\gamma_L(G) \geq 2$. (Or $\gamma_L(G) \geq (n+3)/4$ for $n \geq 5$.)*

Theorem 3.11 *[27] If a graph G with n vertices has a degree sequence $(d_1, d_2, \ldots, d_n)$, with $d_i \geq d_{i+1}$, then $\gamma_L(G) \geq \min\{k : k + (d_1 + d_2 + \cdots + d_k) \geq n\}$.*

Theorem 3.12 *[27] If a graph G with n vertices has a degree sequence $(d_1, d_2, \ldots, d_n)$, with $d_i \geq d_{i+1}$, then $\gamma_L(G) \geq \min\{k : 3k + (d_1 + d_2 + \cdots + d_k)/2 \geq n\}$.*

A graph G is *planar* if G can be drawn on a plane with no crossing edges.

A graph is *outerplanar* if G is planar, and G can be drawn such that every vertex of G lies on the boundary of the region of the plane exterior to G.

Theorem 3.13 *[25] If G is a planar graph with n vertices, then $n \leq 7\gamma_L(G) - 10$ for $\gamma_L(G) \geq 4$. (Or $\gamma_L(G) \geq (n+10)/7$.) Furthermore, this bound is sharp.*

Theorem 3.14 *[25] If G is an outerplanar graph with n vertices, then $n \leq (7\gamma_L(G) - 3)/2$. (Or $\gamma_L(G) \geq (2n+3)/7$.) Furthermore, this bound is sharp.*

The next theorems require more definitions. The *girth* of a graph G, denoted $g(G)$, is the order of the smallest cycle C_g that is a subgraph of G.

Given a graph $G = (V, E)$, a set of vertices $I \subset V$ is *independent* if there are no edges between any pair of vertices in I.

A graph G is *well-covered* if every maximal independent set of vertices of G is a maximum.

A *pendant edge* of a graph is a vertex of degree one, together with the edge that the vertex is incident with. Given a graph G, the *corona* of G, denoted $\mathrm{Cor}(G)$, is the graph formed by attaching a pendant edge to every vertex of G.

Theorem 3.15 *[9] If a graph G has girth $g(G) \geq 5$, then G is well-covered if and only if every independent dominating set of G is also a locating-dominating set of G.*

Theorem 3.16 *[9] Let G be a graph where every dominating set is an LD-set. Then G is a corona, K_1, C_5, or WL_8 (see Figure 5).*

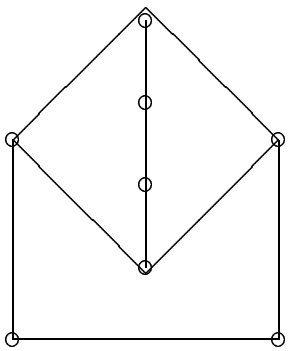


Figure 5: WL_8.

Our final known fact considers the complexity of finding γ_L for a given graph G.

LOCATING-DOMINATING SET

INSTANCE: A graph $G = (V, E)$ and a positive integer k.

QUESTION: Does G have a locating-dominating set of cardinality $\leq k$?

Theorem 3.17 *[6] LOCATING-DOMINATING SET is NP-complete.*

However,

Theorem 3.18 *[26] For trees and series-parallel graphs G, $\gamma_L(G)$ can be computed in linear time, as a function of the number of vertices of G.*

Locating-domination has also been studied in [1], [2], [4], [17], [24], and [29].

4 KNOWN RESULTS - UNIQUE DOMINATING SETS

In this section, we briefly list publications with results on unique dominating sets:

- [3]: Trees with unique minimum paired-dominating sets.
- [7]: Unique total domination in graphs.
- [8]: Block graphs with unique minimum dominating sets.
- [10]: Maximum graphs with a unique minimum dominating set.
- [11]: Unique upper domination in graphs.
- [12]: "Cactus" graphs with unique minimum dominating sets.
- [13]: Unique minimum domination in trees.
- [14]: Graphs with unique dominating sets.
- [15]: Unique domination and independent domination in graphs.
- [16]: Unique maximal dominating cycles in 3-regular graphs.
- [18]: Graphs with unique dominating sets.
- [21]: Graphs with unique minimum total dominating sets.
- [23]: Unique domination in cross-product graphs.
- [30]: Unique minimum edge dominating sets.

The bulk of this work was done by Miranca Fischermann, whose Ph.D. dissertation studied various types of unique domination.

5 NEW RESULTS - TREES WITH UNIQUE MINIMUM LOCATING-DOMINATING SETS

We call a tree that has exactly one γ_L-set a *ULD-tree*. We have already seen one ULD-tree: P_5. Trivially, K_1 is also a ULD-tree. We have also seen a tree that is not a ULD-tree: P_6. (See Figures 1 and 2.)

Our research began by writing a computer program (see Appendix C) that found all ULD-trees of order ≤ 15. We used these ULD-trees to develop a construction to build a family of ULD-trees. See Appendix 6 for the number of ULD-trees of order $n \leq 15$, and Appendix 6 for figures of all ULD-trees of order $n \leq 14$. There are 21 ULD-trees of order 15, so figures of these trees are omitted.

We now need more definitions:

Let S be a set of vertices, and let $u \in S$. A vertex v is a *private neighbor* of u (with respect to S) if $N[v] \cap S = \{u\}$. Furthermore, the *private neighbor set of* u, with respect to S, is $pn[u, S] = \{v : N[v] \cap S = \{u\}\}$.

A vertex v of a tree T is a *leaf* of T if $deg(v) = 1$. v is a *support vertex* of T if v is adjacent to a leaf of T. v is a *strong support vertex* of T if v is adjacent to two or more leaves of T.

If T is a rooted tree, and w is a vertex of T, then T_w denotes the subtree induced by w and all of its descendants. Also, $T - T_w$ denotes the subtree induced by $T - V(T_w)$.

5.1 Properties of ULD-trees

We first present some elementary facts concerning ULD-trees.

Lemma 1 *If T is a ULD-tree with $\gamma_L(T)$-set S and v is a support vertex of T, then $v \in S$.*

Proof. Let $v_1, v_2, \ldots, v_s$ be the leaves adjacent to v, and $u_1, u_2, \ldots, u_t$ be the non-leaf neighbors of v that are not in S. Suppose (to the contrary) that $v \notin S$. Then $\{v_i \mid 1 \leq i \leq s\} \subset S$. Consider $R = S \cup \{v\} - \{v_1\}$. If $u_i \notin N(S) - v$, u_i is either in S or adjacent to a vertex in S other than v, or else u_i would not be dominated in T. Thus R is a locating-dominating set of T. Since $|R| = |S|$, R is a $\gamma_L(T)$-set, and $R \neq S$, contradicting that S is unique. Thus, $v \in S$. ■

Lemma 2 *If T is a ULD-tree, then T has no strong support vertices.*

Proof. Let S be the unique $\gamma_L(T)$-set. Suppose (to the contrary) that T has a strong support vertex v, and that the leaves adjacent to v are $v_1, v_2, \ldots, v_s$, with $s \geq 2$. By Lemma 1, $v \in S$. For S to be a γ_L-set, exactly $s-1$ of the leaves adjacent to v is in S. Without loss of generality, let $v_1 \notin S$. Then $(S - \{v_2\}) \cup \{v_1\}$ is also a $\gamma_L(T)$-set, contradicting that S is unique. Thus, any support vertex of T is adjacent to exactly one leaf. ■

Lemma 3 *If T is a ULD-tree with $\gamma_L(T)$-set S, and v is a leaf of T, then $v \notin S$.*

Proof. Let w be the support vertex adjacent to v. By Lemmas 1 and 2, $w \in S$ and v is the only leaf neighbor of w. Assume (to the contrary) that $v \in S(T)$. There can be at most one vertex u such that $N_S(u) = \{w\}$; otherwise, S is not an LD(T)-set. If no such vertex u exists, then $S - \{v\}$ is an LD(T)-set, contradicting the minimality of S; else $(S \cup \{u\}) - \{v\}$ is a $\gamma_L(T)$-set, contradicting that S is unique. Thus, $v \notin S$.

■

Lemma 4 *If T is a nontrivial ULD-tree, then $diam(T) \geq 4$.*

Proof. If $diam(T) = 1$, then $T = K_2$, which is not a ULD-tree. If $diam(T) = 2$, then T is a star with a strong support vertex, and by Lemma 2, T is not a ULD-tree. If $diam(T) = 3$, then T is a double star that either has a strong support vertex or $T = P_4$. Since P_4 is not a ULD-tree, Lemma 2 implies that T is not a ULD-tree. ■

Observation 5 *If S is a $\gamma_L(T)$-set, and $w \in S$, then $V(T_w) \cap S$ is a $\gamma_L(T_w)$-set.*

5.2 Construction of $\mathcal{T}$

Let $\mathcal{T}$ be a family of trees that contains a labeled $P_5 = (b_1, a_1, b_2, a_2, b_3)$, and let $A(P_5) = \{a_1, a_2\}$, and $B(P_5) = \{b_1, b_2, b_3\}$. The following five operations extend $\mathcal{T}$ by attaching a path to a tree T' in $\mathcal{T}$ to form another tree T, which is then also in $\mathcal{T}$.

- **Operation $\mathcal{T}_1$:** Let v be a vertex in $A(T')$ such that v has a private neighbor v' with respect to $A(T')$. Attach to v the vertex b_1 of a path $P_3 = (b_1, a_1, b_2)$. Let $A(T) = A(T') \cup \{a_1\}$ and $B(T) = B(T') \cup \{b_1, b_2\}$.

- **Operation $\mathcal{T}_2$:** Attach to a support vertex v of T' the vertex a_1 of a path $P_5 = (b_1, a_1, b_2, a_2, b_3)$. Let $A(T) = A(T') \cup \{a_1, a_2\}$ and $B(T) = B(T') \cup \{b_1, b_2, b_3\}$.

- **Operation $\mathcal{T}_3$:** Attach to a vertex v in $B(T')$ the vertex b_1 of a path $P_5 = (b_1, a_1, b_2, a_2, b_3)$. Let $A(T) = A(T') \cup \{a_1, a_2\}$ and $B(T) = B(T') \cup \{b_1, b_2, b_3\}$.

- **Operation $\mathcal{T}_4$:** Attach to a vertex v in $B(T')$ the vertex b_2 of a path $P_5 = (b_1, a_2, b_2, a_2, b_3)$. Let $A(T) = A(T') \cup \{a_1, a_2\}$ and $B(T) = B(T') \cup \{b_1, b_2, b_3\}$.

- **Operation $\mathcal{T}_5$:** Let G_5 be the graph shown in Figure 6. Attach to a support vertex v of T' the vertex b_1 of G_5. Let $A(T) = A(T') \cup \{a_1, a_2, a_3\}$ and $B(T) = B(T') \cup \{b_1, b_2, b_3, b_4, b_5\}$.

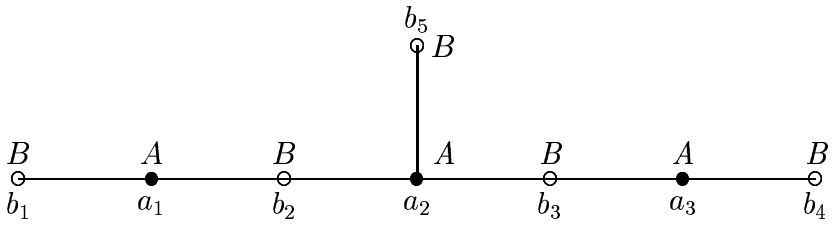


Figure 6: G_5.

Note that P_5 is a ULD-tree, and $A(P_5)$ is the unique $\gamma_L(P_5)$-set.

The five operations are illustrated in Figure 7.

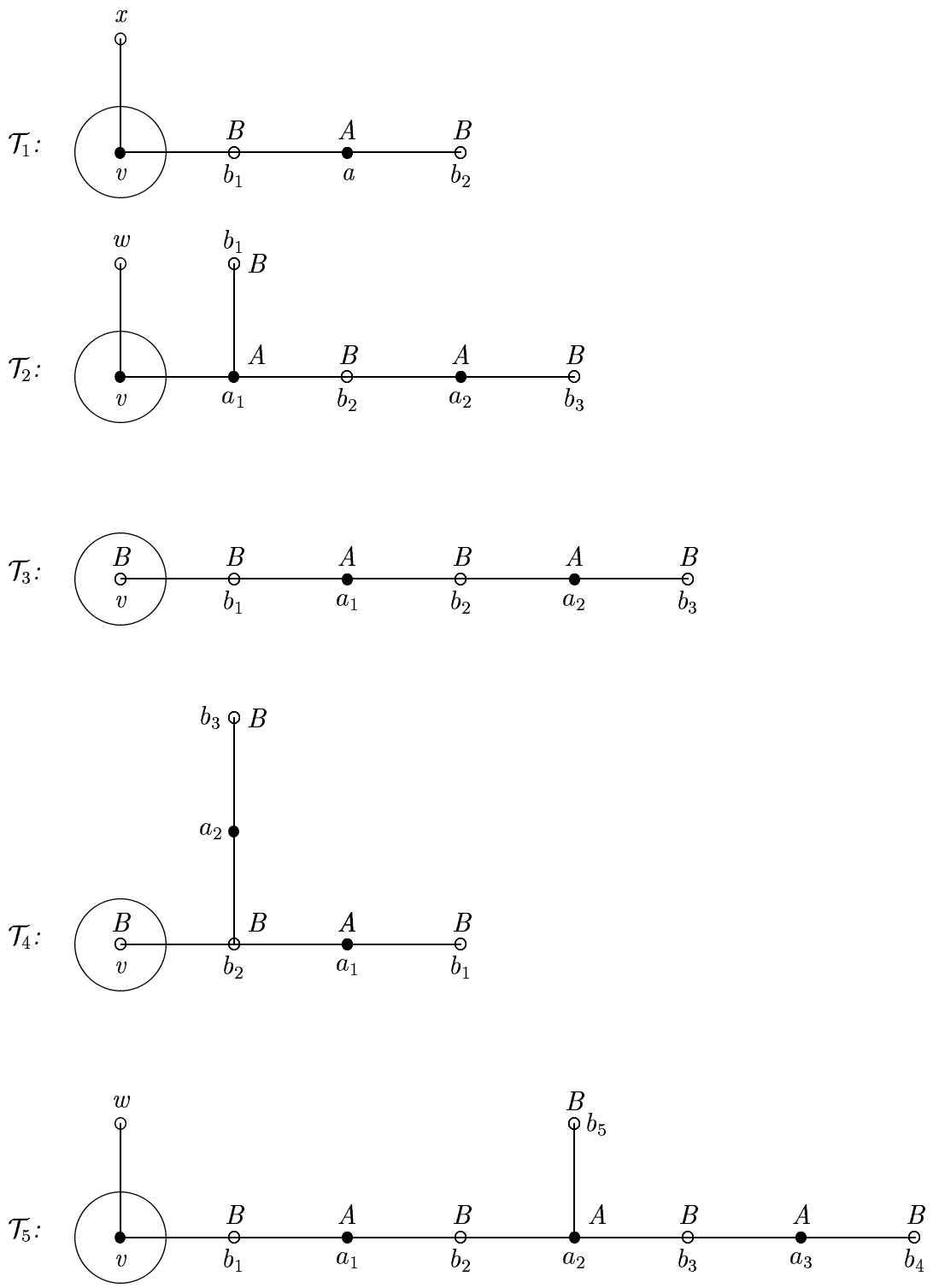


Figure 7: The five $\mathcal{T}_i$ operations. w is a leaf, and x is a private neighbor of v.

5.3 Proof of Construction

This section shows that all trees in $\mathcal{T}$ are ULD-trees.

Theorem 5.1 *If $T \in \mathcal{T}$, then T is a ULD-tree with $\gamma_L(T)$-set $A(T)$.*

Proof. We proceed by induction on the number $s(T)$ of operations required to construct the tree $T \in \mathcal{T}$. If $s(T) = 0$, then $T = P_5$ and T is a ULD-tree with $\gamma_L(T)$-set $A(T)$. This establishes the base case. Assume, then, that $k \geq 1$ is an integer and that each tree $T' \in \mathcal{T}$ with $s(T') < k$ is a ULD-tree with $\gamma_L(T)$-set $A(T)$. Let $T \in \mathcal{T}$ be a tree with $s(T) = k$. Then, T can be obtained from a tree $T' \in \mathcal{T}$ with $s(T') < k$ by one of the operations $\mathcal{T}_1$, $\mathcal{T}_2$ $\mathcal{T}_3$, $\mathcal{T}_4$, and $\mathcal{T}_5$. Applying the inductive hypothesis to the tree T', T' is a ULD-tree with $\gamma_L(T')$-set $A(T')$. Let $S' = A(T')$. We now consider five possibilities depending upon the operation used to obtain T from T'.

Case 1. T is obtained from T' by operation $\mathcal{T}_1$. Suppose T is obtained from T' by adding a path $P_3 = (b_1, a_1, b_2)$ and joining b_1 to a vertex v in T' with an edge where v is a vertex that has a private neighbor, say v', with respect to $A(T')$. Let $S = A(T) = S' \cup \{a_1\}$. Then S is an LDS of T, so $\gamma_L(T) \leq \gamma_L(T') + 1$.

Suppose there exists an LDS D of T with $|D| < \gamma_L(T') + 1$. Let $D' = D \cap V(T')$. Suppose $v \in D$. Since at least one vertex of $\{b_1, a_1, b_2\}$ is in D, $|D'| < \gamma_L(T')$. Since $v \in D'$, D' is an LDS of T' with cardinality less than $\gamma_L(T')$, a contradiction. Suppose $v \notin D$. Then at least two vertices in $\{b_1, a_1, b_2\}$ must be in D, and $|D'| < \gamma_L(T') - 1$. But then $D' \cup \{v\}$ is an LDS of T' with cardinality less that $\gamma_L(T')$, another contradiction. Thus $\gamma_L(T) \geq \gamma_L(T') + 1$, and hence $\gamma_L(T) = \gamma_L(T') + 1$.

We next show that $\gamma_L(T' - \{v\}) \geq \gamma_L(T')$. Suppose (to the contrary) that X' is an LDS of $T' - \{v\}$ and that $|X'| < \gamma_L(T')$. But then $X' \cup \{v\}$ is a $\gamma_L(T')$-set where

v has no private neighbor in T', contradicting the choice of v in Operation $\mathcal{T}_1$. Hence $\gamma_L(T' - \{v\}) \geq \gamma_L(T')$. This implies that $D' = A(T') = S'$.

Let F be a $\gamma_L(T)$-set and $F' = F \cap V(T')$. By our above argument, $|F'| \geq \gamma_L(T')$, so exactly one vertex from $\{b_1, a_1, b_2\}$ is in F. For $\{a_1, b_2\}$ to be dominated, at least one of them is in F. If b_2 is in F, then v is in F, but then $F' = S'$, and thus $N_S(b_1) = N_S(v') = \{v\}$, contradicting that F is locating. Thus, $a \in F$, $F' = S'$, and T is a ULD-tree with $\gamma_L(T)$-set $F = A(T)$.

Case 2. T is obtained from T′ by operation $\mathcal{T}_2$. Suppose T is obtained from T' by adding the path $P_5 = (b_1, a_1, b_2, a_2, b_3)$ and the edge a_1v where v is a support vertex of T'. By Lemma 1, $v \in S'$ and v' is the only leaf neighbor of v. Since $S' \cup \{a_1, a_2\}$ is an LDS of T, we have $\gamma_L(T) \leq \gamma_L(T') + 2$.

Now let D be a $\gamma_L(T)$-set. Using an argument similar to Case 1, it is straightforward to see that $D' = D \cap V(T')$ is the unique $\gamma_L(T')$-set $S' = A(T')$. Also, exactly two vertices from a_1, b_1, b_2, a_2, and b_3 are in D. Since $v \in D$, the only two of these vertices that locate-dominate the attached P_5 are a_1 and a_2. Again we have that T is a ULD-tree with the unique $\gamma_L(T)$-set $A(T)$.

Case 3. T is obtained from T′ by operation $\mathcal{T}_3$. Suppose T is obtained from T' by adding the path $P_5 = (b_1, a_1, b_2, a_2, b_3)$ and the edge b_1v where $v \in B(T')$.

Let D be a $\gamma_L(T)$-set. We first show that $\gamma_L(T) = \gamma_L(T') + 2$ and that $D' = D \cap V(T')$ is the unique $\gamma_L(T')$-set $A(T')$. The $\gamma_L(T')$-set $A(T')$ can be extended to a locating-dominating set of T by adding to it the vertices a_1 and a_2, and so $\gamma_L(T) \leq \gamma_L(T') + 2$. On the other hand, at least two vertices from $\{a_1, a_2, b_2, b_3\}$ are in D.

If D' is not an LDS of T', then D' is an LDS of $T' - \{v\}$, implying $b_1 \in D$ to locate-dominate v. Thus $|D'| \leq \gamma_L(T) - 3$. Now $D' \cup \{v\}$ is an LDS of T' and $\gamma_L(T) - 2 \leq \gamma_L(T') \leq |D' \cup \{v\}| \leq \gamma_L(T) - 3 + 1$. Hence equality follows and $|D' \cup \{v\}| = \gamma_L(T')$, contradicting that $A(T')$ is the unique $\gamma_L(T')$-set. Thus D' is an LDS of T', and $\gamma_L(T') \leq |D'| \leq \gamma_L(T) - 2$. Hence $\gamma_L(T) = \gamma_L(T') + 2$, and $|D'| = \gamma_L(T)$. Therefore, D' is the unique $\gamma_L(T')$-set $A(T')$, and exactly two vertices from $\{a_1, a_2, b_2, b_3\}$ are in D. Note that $v \notin D$.

It is now easily checked that $D = D' \cup \{a_1, a_2\} = A(T') \cup \{a_1, a_2\}$. Hence, T is a ULD-tree with $\gamma_L(T)$-set $A(T)$.

Case 4. T is obtained from T′ by operation $\mathcal{T}_4$.

The proof for this case is similar for the proof of Case 3, so it is omitted.

Case 5. T is obtained from T′ by operation $\mathcal{T}_5$.

Suppose T is obtained from T' by adding the graph G_5 (see Figure 6) and attaching vertex b_1 of G_5 to a support vertex v in T'. Since T' is a ULD-tree, Lemma 1 implies that $v \in S'$ and that there is exactly one leaf adjacent to v, say w, and $w \notin S'$. Let $S = S' \cup \{a_1, a_2, a_3\}$. Then S is an LDS of T, so $\gamma_L(T) \leq \gamma_L(T') + 3$.

Note that every $\gamma_L(T)$-set must contain one of v and w to dominate w. Hence for any $\gamma_L(T)$-set D, it follows that $D' = D \cap V(T')$ is an LDS of T'. Therefore, $|D'| \geq \gamma_L(T')$. Moreover, at least one of a_2 and b_5 is in D to dominate b_5; at least one of b_1, a_1, and b_2 is in D to dominate a_1, and at least one of b_3, a_3, and b_4 is in D to dominate a_3. Thus, $\gamma_L(T) \geq |D'| + 3$. Hence, $\gamma_L(T) = \gamma_L(T') + 3$ and D' is a $\gamma_L(T')$-set. Then D' is the unique $\gamma_L(T')$-set $A(T')$, $v \in D'$, and exactly three of $b_1, a_1, b_2, a_2, b_3, a_3, b_4$, and b_5 (the vertices of G_5) are in D. Since $v \in D$ and $N(w) \cap D = \{v\}$, to locate-dominate the vertices in G_5, $a_1, a_2, a_3 \in D$. Thus,

$D = D' \cup \{a_1, a_2, a_3\}$ is the unique $\gamma_L(T)$-set and hence, T is a ULD-tree with $\gamma_L(T)$-set $A(T)$. ■

Observation 6 *If T is a ULD-tree, and $n(T) \leq 15$, then $\gamma_L(T) = \lfloor n(T)/3 \rfloor + 1$. (This should hold for $n(T) \in \{16, 17\}$, but will not hold for $n(T) \geq 18$.)*

The obvious future work would be to attempt to characterize ULD-trees.

Another extension of this work is from locating-domination to *identifying codes*, also called *differentiating-domination*. Recall that for a graph G, a subset of vertices S is a locating-dominating set if for every pair of vertices u, v in $V(G) - S$, $N_S(u)$ and $N_S(v)$ are both nonempty, and $N_S(u) \neq N_S(v)$.

For a set $S \subset V(G)$ to be an identifying code, we simply allow vertices u, v to also be in S, and then make the same requirement: $N_S(u)$ and $N_S(v)$ are both nonempty, and $N_S(u) \neq N_S(v)$. Then the *identifying code number* of G, denoted $IC(G)$, is the minimum cardinality of a identifying code of G, if such a set exists. For many graphs, an identifying code does *not* exist; for example, it does not exist for any complete graph other than the trivial K_1.

An open problem is to characterize trees with unique minimum identifying codes, but this task seems to be even more difficult than characterizing ULD-trees.

We did show that unlike most graph parameters which are monotone increasing with respect to the number of vertices in a graph, the identifying code number is not. In particular, the identifying code number can and does decrease in *Queen's graphs*. In particular, we found via a computer program that the identifying code number of the 3×3 Queen's graph is 6, while the identifying code number of the 4×4 Queen's graph is 5. We would like to both prove our program's results, and extend them to larger Queen's graphs.

It would also be interesting to study the locating-domination number of unstudied families of graphs, such as *grid graphs*.

BIBLIOGRAPHY

[1] N. Bertrand, I. Charon, O. Hudry, and A. Lobstein, Identifying and locating-dominating codes on chains and cycles. *European J. Combin.* **25** (2004), no. 7, 969–987.

[2] D. I. Carson, On generalized locating-domination. *Graph theory, combinatorics, and algorithms, Vol.* **1**, **2** *(Kalamazoo, MI, 1992),* 161–179, Wiley, New York, 1995.

[3] M. Chellali and T. Haynes, Trees with unique minimum paired-dominating sets. *Ars Combin.* **25** (2004), 3–12.

[4] I. Charon, O. Hudry, and A. Lobstein, Minimizing the size of an identifying or locating-dominating code in a graph is NP-hard. *Theoret. Comput. Sci.* **290** (2003), no. 3, 2109–2120.

[5] G. Chartrand and L. Lesniak, *Graphs & Digraphs*, 3rd ed., Chapman & Hall, New York, 1996.

[6] C. J. Colbourn, P. J. Slater, and L. K. Stewart, Locating-dominating sets in series-parallel networks. *Congr. Numer.* **56** (1987), 135–162.

[7] M. Fischermann, Unique total domination graphs. *Ars Combin.* **73** (2004), 289–297.

[8] M. Fischermann, Block graphs with unique minimum dominating sets. *Discrete Math.* **240** (2001), 247–251.

[9] A. Finbow and B. L. Hartnell, On locating dominating sets and well-covered graphs. *Congr. Numer.* **65** (1988), 191–200.

[10] M. Fischermann, D. Rautenbach, and L. Volkmann, Maximum graphs with a unique minimum dominating set. *Discrete Math.* **260** (2003), 197–203.

[11] M. Fischermann and L. Volkmann, Unique independence, upper domination and upper irredundance in graphs. *J. Combin. Math. Combin. Comput.* **47** (2003), 237–249.

[12] M. Fischermann and L. Volkmann, Cactus graphs with unique minimum dominating sets. *Util. Math.* **63** (2003), 229–238.

[13] M. Fischermann and L. Volkmann, Unique minimum domination in trees. *Australas. J. Combin.* **25** (2002), 117–124.

[14] M. Fischermann and L. Volkmann, Graphs with unique dominating sets. *Electron. Notes Discrete Math.* **5** (2000), 3 pp. (electronic)

[15] M. Fischermann, L. Volkmann, and I. Zverovich, Unique irredundance, domination, and independent domination in graphs. *Discrete Math.* **305** (2005), 190–200.

[16] H. Fleischner, Uniqueness of maximal dominating cycles in 3-regular graphs and of Hamiltonian cycles in 4-regular graphs. *J. Graph Theory* **18** (1994), 449-459.

[17] J. Gimbel, B. van Gorden, M. Nicolescu, C. Umstead, and N. Vaianna, Location with dominating sets. *Congr. Numer.* **151** (2001), 129–144.

[18] G. Gunther, B. Hartnell, L. R. Markus, and D. Rall, Graphs with unique minimum dominating sets. *Congr. Numer.* **101** (1994), 55–63.

[19] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, *Fundamentals of Domination in Graphs*, Marcel Dekker, New York, 1998.

[20] T. W. Haynes, S. T. Hedetniemi, and P. J. Slater (eds), *Domination in Graphs: Advanced Topics*, Marcel Dekker, New York, 1998.

[21] T. W. Haynes and M. A. Henning, Trees with unique minimum total dominating sets. *Discuss. Math.-Graph Theory* **22** (2002), 233–246.

[22] M. A. Henning and O. R. Oellerman, Metric-locating-dominating sets in graphs. *Ars Combin.* **73** (2004), 129–141.

[23] J. D. Masters, Q. F. Stout, and D. M. Van Wieren, Unique domination in cross-product graphs. *Congr. Numer.* **118** (1996), 49–71.

[24] N. Megiddo, E. Zemel, and S. L. Hakimi, The maximum coverage location problem, *SIAM J. Alg. Disc. Meth.* **4** (1983), 253–261.

[25] D. F. Rall and P. J. Slater, On location-domination numbers for certain classes of graphs. *Congr. Numer.* **45** (1984), 97–106.

[26] P. J. Slater, Domination and location in acyclic graphs. *Networks* **17** (1987), no. 1, 55–64.

[27] P. J. Slater, Locating dominating sets and locating-dominating sets. *Graph theory, combinatorics, and algorithms, Vol.* **1**, **2** *(Kalamazoo, MI, 1992),* 1073–1079, Wiley, New York, 1995.

[28] P. J. Slater, Dominating and reference sets in a graph. *J. Math. Phys. Sci.* **22** (1988), no. 4, 445–455.

[29] P. J. Slater, Fault-tolerant locating-dominating sets. *Discrete Math.* **249** (2002), no. 1–3, 179–189.

[30] J. Topp, Graphs with unique minimum edge dominating sets and graphs with unique maximum independent sets of vertices. *Discrete Math.* **121** (1993), no. 1–3, 199–210.

APPENDICES

Appendix A: Number of ULD-trees of order n.

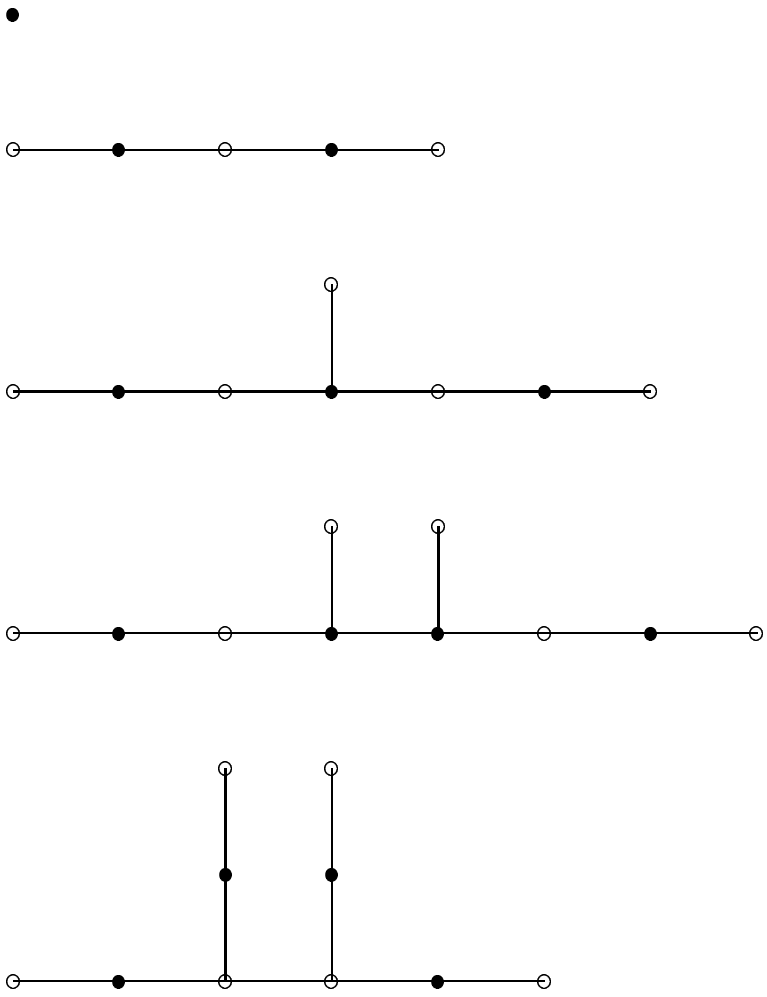
Table 1: Number of ULD-trees of order n

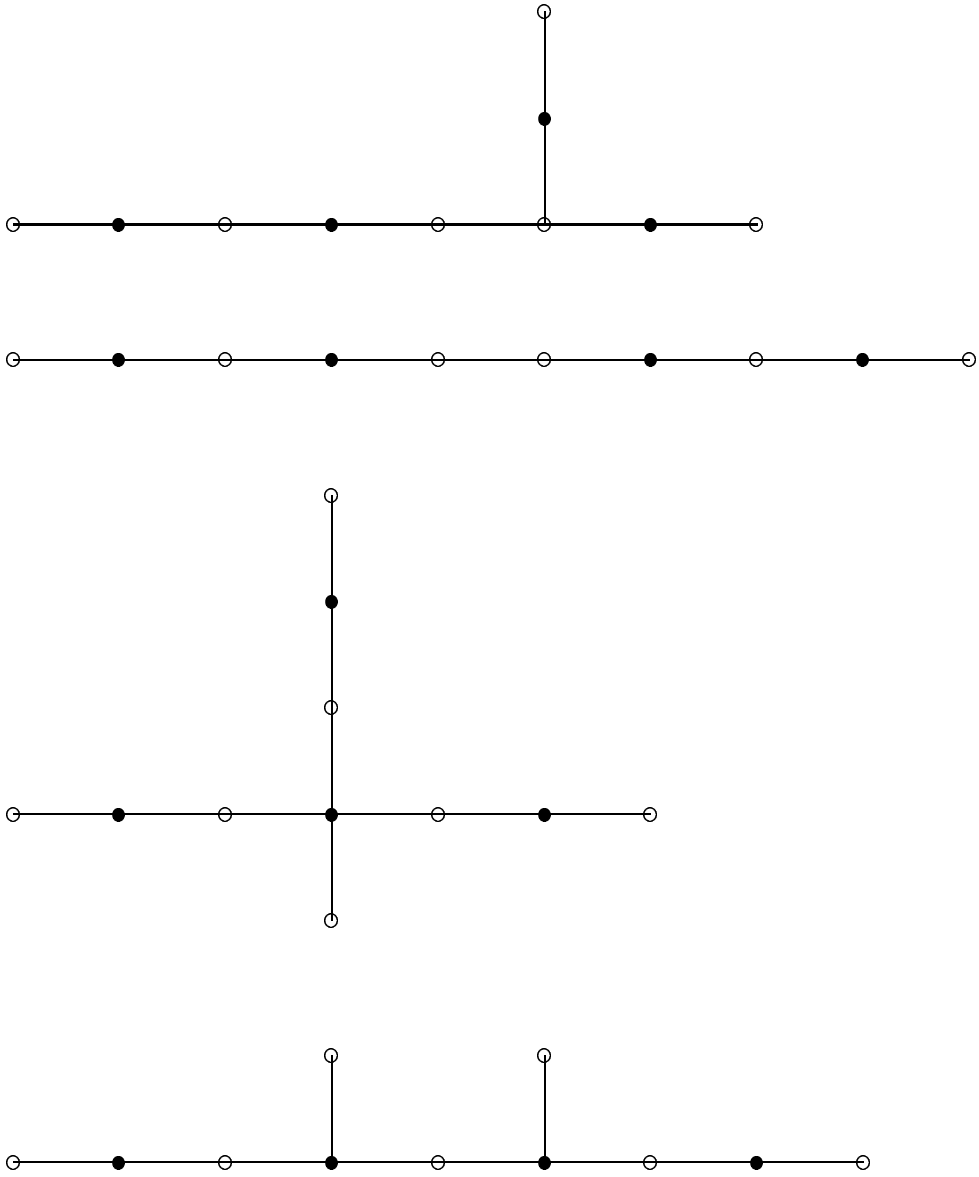
$n(T)$	Number of ULD-trees	$\gamma_L(T)$
1	1	1
2	0	-
3	0	-
4	0	-
5	1	2
6	0	-
7	0	-
8	1	3
9	0	-
10	4	4
11	2	4
12	0	-
13	9	5
14	3	5
15	21	6

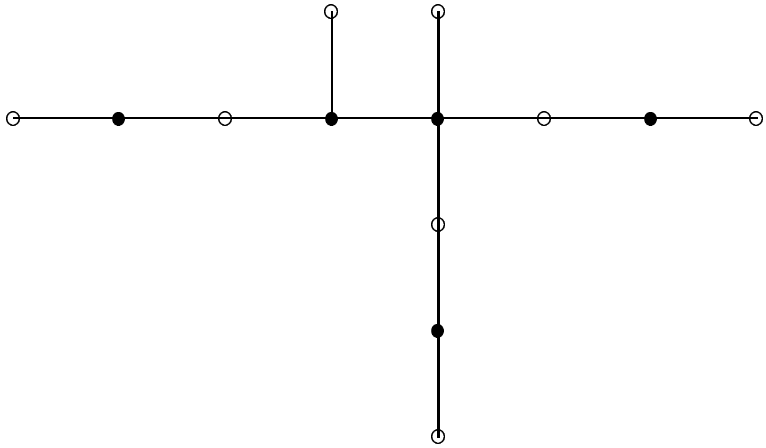
APPENDICES

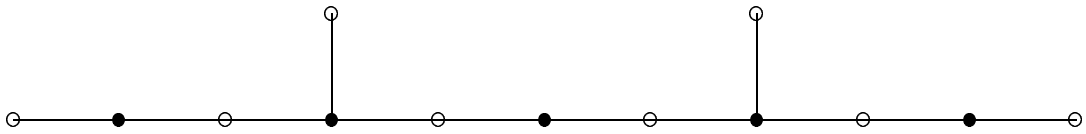
Appendix B: All ULD-trees of Order ≤ 14

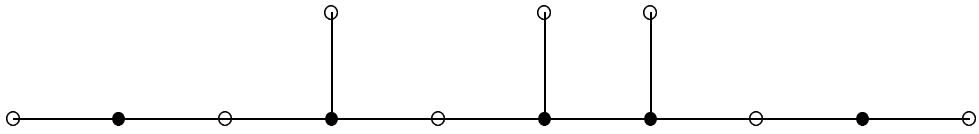
Following are figures of all ULD-trees of order ≤ 14. The shaded vertices form each tree's unique minimum locating-dominating set.

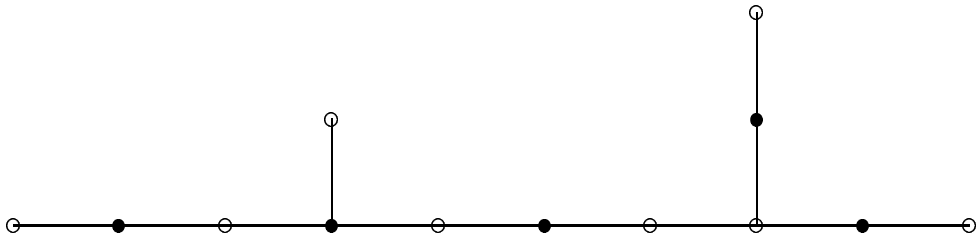


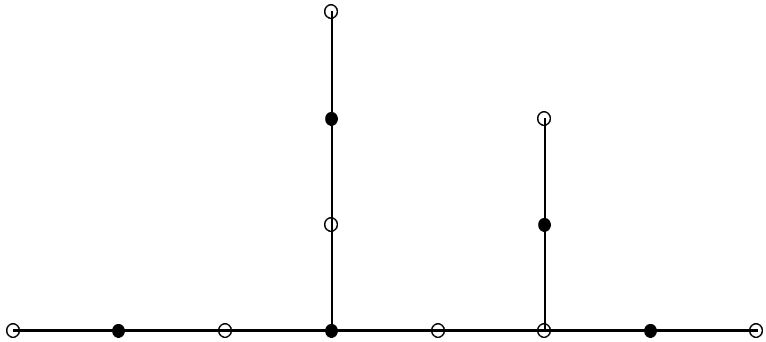


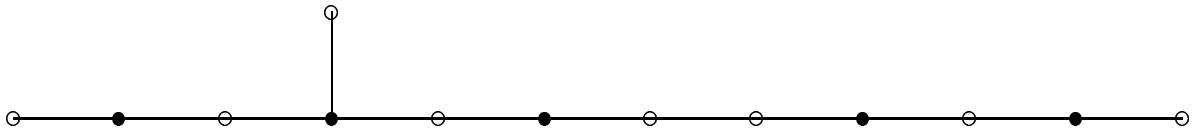


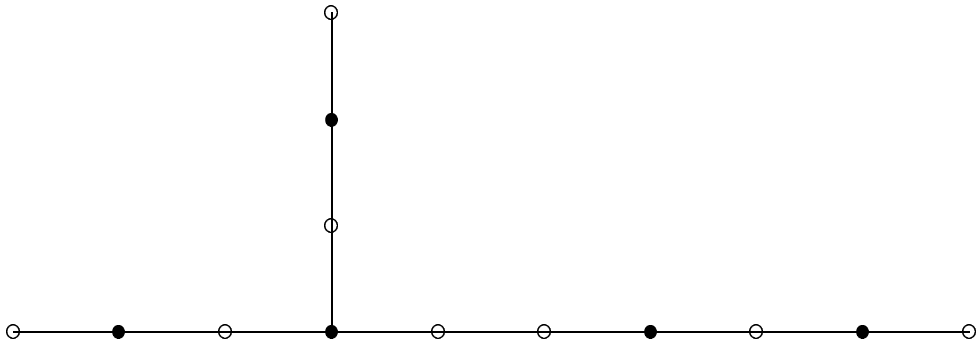


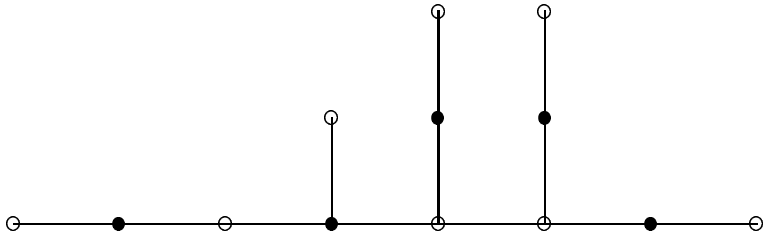


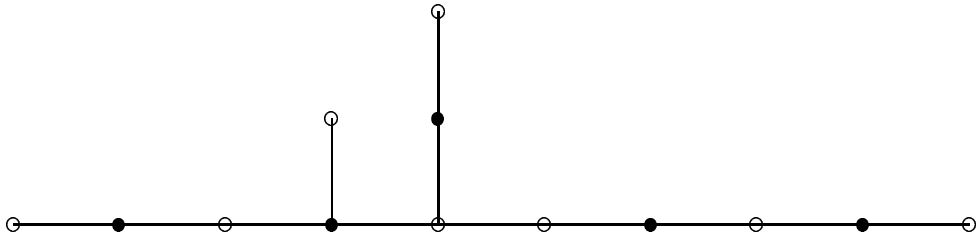


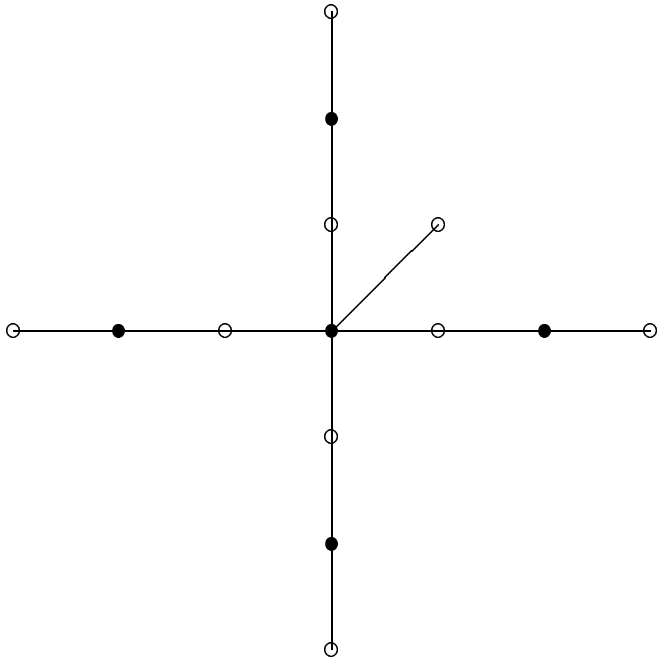


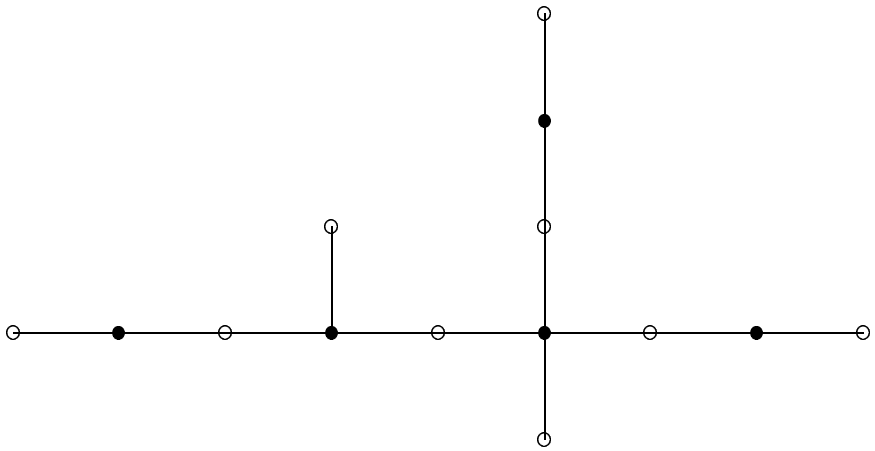


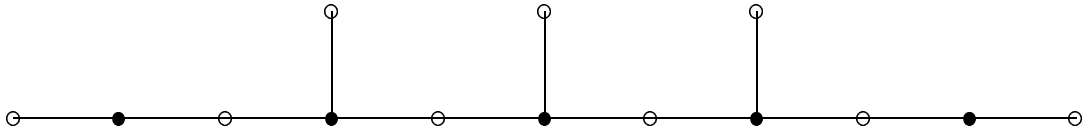












Appendix C: Programs and Modules

Following are selected programs and modules used in this research. The programs and modules are written in Perl.

Graph.pm: a module implenting a graph as an object.

```
# created: sml/2005-02-03
# updated: sml/2006-03-25
# Graph.pm - a graph.
use strict;
use warnings;
use lib '/home/sml/lib';

package Graph;
my $CLASS = 'Graph';

use Object;
our @ISA = qw(Object);

use SML qw(subset_generator list_intersection list_minus list_union uniq min max);

### CLASS METHODS

sub new {
  my $class = shift;
  my $o = {
    # name, a string, optional.
    name => '',

    # edge set.  edges are represented by /\w-\w/.
    # isolates are represented with /\w/.
    E => [],

    # vertex set.  a list of /\w/.
    V => [],

    # degree hash.
    deg => {},
    delta_small => undef,  # min deg.
    delta_large => undef,  # max deg.

    # open neighborhood hash, per vertex.
    No => {},

    # eccentricity params.  only connected graphs have these.
    ecc => undef,    # {}, vertices => eccentricity
    diam => undef,   # diameter
    rad => undef,    # radius
    center => undef, # [], vertices in center of graph

    # domination numbers.  all sets are minimum.
    gamma => undef,
    gamma_set => [],
    gamma_L => undef,
    gamma_L_set => [],
    gamma_D => undef,
```

```
    gamma_D_set => [],

    # changing/unchanging stuff.
    vertex_removal_gamma_diff => {},
    edge_removal_gamma_diff => {},
    edge_addition_gamma_diff => {},
    gamma_changing_unchanging_region => undef,

    @_,
  };
  bless $o, $class;
  $o->init;
  return $o;
}

sub queens_graph_edges {
# returns the edge set of an nxn queens' graph.
  my $class = shift;
  my $n = shift;

  my @row = ('a' .. chr(ord('a') + $n - 1));
  my @col = (1 .. $n);

  my @e;

  for my $ri (0 .. $#row) {
    for my $ci (0 .. $#col) {
      my $r = $row[$ri];
      my $c = $col[$ci];
      my $v = "$r$c";

      # $v is adjacent to all vertices in its row.
      for my $r1 (@row) {
        next if $r1 eq $r;
        push @e, "$v-$r1$c";
      }

      # $v is adjacent to all vertices in its col.
      for my $c1 (@col) {
        next if $c1 eq $c;
        push @e, "$v-$r$c1";
      }

      my($ri2, $ci2);

      # $v is adjacent to all vertices in its main diag.
      $ri2 = $ri; $ci2 = $ci;
      {
        $ri2++;  last if $ri2 >= $n;
        $ci2++;  last if $ci2 >= $n;
        push @e, "$v-$row[$ri2]$col[$ci2]";
        redo;
      }
      $ri2 = $ri; $ci2 = $ci;
      {
        $ri2--;  last if $ri2 < 0;
        $ci2--;  last if $ci2 < 0;
        push @e, "$v-$row[$ri2]$col[$ci2]";
        redo;
      }
```

```
      # $v is adjacent to all vertices in its reverse diag.
      $ri2 = $ri; $ci2 = $ci;
      {
        $ri2--;  last if $ri2 < 0;
        $ci2++;  last if $ci2 >= $n;
        push @e, "$v-$row[$ri2]$col[$ci2]";
        redo;
      }
      $ri2 = $ri; $ci2 = $ci;
      {
        $ri2++;  last if $ri2 >= $n;
        $ci2--;  last if $ci2 < 0;
        push @e, "$v-$row[$ri2]$col[$ci2]";
        redo;
      }

      # if @e is empty, we have Qu1.
      if (!@e) { @e = 'a1' }
    }
  }

  return @e;
}

sub grid_graph_edges {
# returns the edge set of an r x c grid graph.
  my $class = shift;
  my $rows = shift;
  my $cols = shift;

  my @row = ('a' .. chr(ord('a') + $rows - 1));
  my @col = (1 .. $cols);

  my @e;

  for my $ri (0 .. $#row) {
    for my $ci (0 .. $#col) {
      my $r = $row[$ri];
      my $c = $col[$ci];
      my $v = "$r$c";

      # add edge above, if not on top.
      push @e, "$v-$row[$ri - 1]$c"
        unless $ri == 0;

      # add edge to left, if not at left.
      push @e, "$v-$r$col[$ci - 1]"
        unless $ci == 0;
    }
  }

  return @e;
}

### OBJECT METHODS

sub info {
# returns a string with what we know about a graph.
  my $o = shift;
  my $name = $o->name;
  my @E = $o->E;
  my $m = $o->m;
```

```
  my @V = $o->V;
  my $n = @V;

  my $info = <<EOF;
name($name) n($n) m($m) V(@V) E(@E)
EOF

  return $info;
}

sub m {
  my $o = shift;
  return $o->{m} if exists $o->{m};
  my $m = $o->size;
  $o->{m} = $m;
  return $m;
}

sub n {
  my $o = shift;
  return $o->{n} if exists $o->{n};
  my @V = $o->V;
  my $n = @V;
  $o->{n} = $n;
  return $n;
}


sub size {
  my $o = shift;
  my @E = $o->E;
  my @edges = grep /-/, @E;
  my $size = @edges;
  return $size;
}

sub init {
# get the basic properties of a graph.
  my $o = shift;
  my @E = $o->E;

  # we gotta sort the edges.
  my @E_new;
  for my $e (@E) {
    my $e_new = join '-', sort split /-/, $e;
    push @E_new, $e_new;
  }
  @E_new = uniq(@E_new);
  $o->{E} = [ @E_new ];

  my @vertices = map split(/-/, $_), @E;
  my %V = map { ($_, 1) } @vertices;
  my @V = sort keys %V;
  $o->V( [@V] );

  # get the degree info.
  $o->deg;

  return;
}

sub E {
```

```
# the edge set.  see ->new.
  my $o = shift;
  $o->{E} = shift if @_;
  return @{ $o->{E} };
}

sub V {
# the vertex set.
  my $o = shift;
  $o->{V} = shift if @_;
  return @{ $o->{V} };
}

sub deg {
# the degree hash, vertex => degree.
  my $o = shift;
  my %deg = %{$o->{deg}};
  return %deg if %deg;

  my @E = $o->E;
  my @V = $o->V;
  for my $v (@V) {
    my $deg = grep /^$v-/ || /-$v$/, @E;
    $deg{$v} = $deg;
  }

  $o->{deg} = { %deg };

  $o->{delta_small} = min(values %deg);
  $o->{delta_large} = max(values %deg);

  return %deg;
}

sub No {
# returns the open neighborhood of $v.
  my $o = shift;
  my $v = shift;

  return @{ $o->{No}{$v} } if $o->{No}{$v};

  my @E = $o->E;

  my @No;
  for (@E) {
    if (/^(\w+)-$v$/) {
      push @No, $1;
    }
    elsif (/^$v-(\w+)$/) {
      push @No, $1;
    }
  }

  $o->{No}{$v} = [ @No ];

  return @No;
}

sub Nc {
# returns the closed neighborhood of $v.
  my $o = shift;
  my $v = shift;
```

```
  my @No = $o->No($v);
  my @Nc = ($v, @No);
  return @Nc;
}

sub induced_subgraph {
# given a set of vertices, returns the subgraph induced by
# those vertices, or <@v>.
  my $o = shift;
  my @v = @_;
  my @E = $o->E;
  my @V = $o->V;

  # choose the edges that have both vertices in @v.
  my @e;
  for my $e (@E) {
    my @v_e = split /-/, $e;
    next unless @v_e == 2;
    next unless grep $v_e[0] eq $_, @v;
    next unless grep $v_e[1] eq $_, @v;
    push @e, $e;
  }

  # add in isolates of @v.
  if (!@e) {
    # <@v> is all isolates in @v.
    @e = @v;
  }
  else {
    for my $v (@v) {
      next if grep /^$v-/, @e;
      next if grep /-$v$/, @e;
      push @e, $v;
    }
  }

  my $induced_subgraph = Graph->new(E => [ @e ]);

  return $induced_subgraph;
}

sub is_gamma_set {
# returns true if @gamma_set dominates $o, i.e. if
# every vertex in $o has a (closed) neighbor in @gamma_set.
  my $o = shift;
  my @gamma_set = @_;
  my @V = $o->V;

  for my $v (@V) {
    my @Nc_v = $o->Nc($v);
    my @list_intersection = list_intersection(\@gamma_set, \@Nc_v);
    return 0 unless @list_intersection;
  }

  return 1;
}

sub gamma {
# the domination number.
  my $o = shift;
  my %param = @_;
```

```
  if (defined $o->{gamma}) {
    return($o->{gamma}, $o->{gamma_set});
  }
  # find the domination number.
  my @V = $o->V;
  my $n = @V;
  if ($param{subset}) {
    @V = @{$param{subset}};
  }
  my $subset_generator = subset_generator(\@V);
  my $gamma = 0;
  my @gamma_sets;
  my $count = 0;
  while (my @subset = $subset_generator->()) {
    $count++;
    my $r = @subset;
    # keep track of things if the search takes a long time.
    if ($count % 100_000 == 0) {
      printf "n($n) r($r) count($count)...\n";
    }
    if ($o->is_gamma_set(@subset)) {
      last if $gamma and @subset > $gamma;
      $gamma = @subset;
      push @gamma_sets, [ @subset ];
      last if !exists $param{max};
      last if exists $param{max} and @gamma_sets >= $param{max};
    }
  }
  $o->{gamma} = $gamma;
  $o->{gamma_set} = $gamma_sets[0];
  return wantarray ? ($o->{gamma}, @gamma_sets) : $o->{gamma};
}

sub is_gamma_locating_set {
# returns 1 if the subset is a locating set; 0 otherwise.
# a set S is a locating set if for all u,v in V - S,
# N(u) inter S != N(v) inter S.
  my $o = shift;
  my @locating_set = @_;
  my @V = $o->V;
  # find the S-N(v) intersections.
  my @V_S = list_minus(\@V, \@locating_set);
  my @inters;
  for my $v (@V_S) {
    my @No = $o->No($v);
    my @inter = list_intersection(\@No, \@locating_set);
    @inter = sort @inter;
    push @inters, "@inter";
  }
  my @inters_uniq = uniq(@inters);
  return @inters == @inters_uniq ? 1 : 0;
}

sub gamma_L {
```

```
# the locating domination number.
  my $o = shift;
  my %param = @_;
  if (defined $o->{gamma_L}) {
    return($o->{gamma_L}, $o->{gamma_L_set});
  }

  # find the locating domination number.
  my @V = $o->V;
  if ($param{subset}) {
    @V = @{$param{subset}};
  }
  my $subset_generator = subset_generator(\@V);

  my $gamma_L = 0;
  my @gamma_L_sets;

  while (my @subset = $subset_generator->()) {
    if ($o->is_gamma_set(@subset) and $o->is_gamma_locating_set(@subset)) {
      last if $gamma_L and @subset > $gamma_L;
      $gamma_L = @subset;
      push @gamma_L_sets, [ @subset ];
      last if !exists $param{max};
      last if exists $param{max} and @gamma_L_sets >= $param{max};
    }
  }

  $o->{gamma_L} = $gamma_L;
  $o->{gamma_L_set} = $gamma_L_sets[0];
  return wantarray ? ($o->{gamma_L}, @gamma_L_sets) : $o->{gamma_L};
}

sub is_gamma_differentiating_set {
# returns 1 if the subset is a differentiating dominating set; 0 otherwise.
# a set S is a differentiating set if for all u,v in V,
# N[u] inter S != N[v] inter S.
  my $o = shift;
  my @locating_set = @_;
  my @V = $o->V;

  # find the S-N(v) intersections.
  my @inters;
  for my $v (@V) {
    my @Nc = $o->Nc($v);
    my @inter = list_intersection(\@Nc, \@locating_set);
    @inter = sort @inter;
    push @inters, "@inter";
  }

  my @inters_uniq = uniq(@inters);
  return @inters == @inters_uniq ? 1 : 0;
}

sub gamma_D {
# the differientating domination number.
  my $o = shift;
  my %param = @_;
  if (defined $o->{gamma_D}) {
    return($o->{gamma_D}, $o->{gamma_D_set});
```

```
  }
  # find the differentiating domination number.
  my @V = $o->V;
  if ($param{subset}) {
    @V = @{$param{subset}};
  }
  my $subset_generator = subset_generator(\@V);

  my $gamma_D = 0;
  my @gamma_D_sets;

  while (my @subset = $subset_generator->()) {
    if ($o->is_gamma_set(@subset) and
        $o->is_gamma_differentiating_set(@subset)) {
      last if $gamma_D and @subset > $gamma_D;
      $gamma_D = @subset;
      push @gamma_D_sets, [ @subset ];
      last if !exists $param{max};
      last if exists $param{max} and @gamma_D_sets >= $param{max};
    }
  }

  $o->{gamma_D} = $gamma_D;
  $o->{gamma_D_set} = $gamma_D_sets[0];
  return wantarray ? ($o->{gamma_D}, @gamma_D_sets) : $o->{gamma_D};
}

sub is_gamma_total_set {
# returns true if @gamma_set total-dominates $o, i.e. if
# every vertex in $o has an OPEN neighbor in @gamma_set.
  my $o = shift;
  my @gamma_set = @_;
  my @V = $o->V;

  for my $v (@V) {
    my @No_v = $o->No($v);
    my @list_intersection = list_intersection(\@gamma_set, \@No_v);
    return 0 unless @list_intersection;
  }

  return 1;
}


sub gamma_t {
# the total domination number.
  my $o = shift;
  my %param = @_;
  if (defined $o->{gamma_t}) {
    return($o->{gamma_t}, $o->{gamma_t_set});
  }
  # find the total domination number.
  my @V = $o->V;
  if ($param{subset}) {
    @V = @{$param{subset}};
  }
  my $subset_generator = subset_generator(\@V);

  my $gamma_t = 0;
```

```
  my @gamma_t_sets;

  while (my @subset = $subset_generator->()) {
    if ($o->is_gamma_total_set(@subset)) {
      last if $gamma_t and @subset > $gamma_t;
      $gamma_t = @subset;
      push @gamma_t_sets, [ @subset ];
      last if !exists $param{max};
      last if exists $param{max} and @gamma_t_sets >= $param{max};
    }
  }

  $o->{gamma_t} = $gamma_t;
  $o->{gamma_t_set} = $gamma_t_sets[0];
  return wantarray ? ($o->{gamma_t}, @gamma_t_sets) : $o->{gamma_t};
}

sub gamma_pr {
# the paired domination number.
  my $o = shift;
  my %param = @_;
  if (defined $o->{gamma_pr}) {
    return($o->{gamma_pr}, $o->{gamma_pr_set});
  }

  # find the paired domination number.
  my @V = $o->V;
  if ($param{subset}) {
    @V = @{$param{subset}};
  }
  my $subset_generator = subset_generator(\@V);

  my $gamma_pr = 0;
  my @gamma_pr_sets;

  my $subset_size = 0;
  while (my @subset = $subset_generator->()) {
    if ($subset_size != @subset) {
      $subset_size = @subset;
    }
    if ($o->is_gamma_set(@subset)) {
      my $induced_subgraph = $o->induced_subgraph(@subset);
      if ($induced_subgraph->has_a_perfect_matching) {
        last if $gamma_pr and @subset > $gamma_pr;
        $gamma_pr = @subset;
        push @gamma_pr_sets, [ sort @subset ];
        last if !exists $param{max};
        last if exists $param{max} and @gamma_pr_sets >= $param{max};
      }
    }
  }

  $o->{gamma_pr} = $gamma_pr;
  $o->{gamma_pr_set} = $gamma_pr_sets[0];
  return wantarray ? ($o->{gamma_pr}, @gamma_pr_sets) : $o->{gamma_pr};
}

sub has_a_perfect_matching {
```

```
# returns 1 if the graph has a perfect matching;
# if the graph has an independent set of edges that
# covers all vertices.
  my $o = shift;
  my @E = $o->E;
  my @V = $o->V;

  # a graph can't have a perfect matching if it has isolates.
  return 0 if grep !/-/, @E;

  # and it can't if it has an odd number of vertices.
  return 0 if @V % 2 == 1;

  # start finding subsets of the edges.
  my $subset_generator = subset_generator(\@E, reverse => 1);
  while (my @subset = $subset_generator->()) {
    # the subset must cover all vertices.
    next unless 2 * @subset == @V;
    last if 2 * @subset < @V;

    # require that the subset has no duplicated vertices;
    # then the edges are independent.
    my @v = map split(/-/, $_), @subset;
    my @v_uniq = uniq(@v);
    return 1 if @v == @v_uniq;
  }

  # no dice.
  return 0;
}

sub edge_removal_gamma_diff {
# returns a hash with edge ab as key and
# gamma(G - ab) - gamma(G) as value.  the value
# is either 0 or 1.
  my $o = shift;
  return %{$o->{edge_removal_gamma_diff}} if %{$o->{edge_removal_gamma_diff}};

  my $gamma = $o->gamma;
  my %edge_removal_gamma_diff;
  my @E = $o->E;
  for my $e (@E) {
    my @e_v = split /-/, $e;
    next unless @e_v == 2;  # we want edges, not isolates.
    my @E2 = grep $_ ne $e, @E;

    # if we just removed a vertex, we need to put it back.
    for my $v (@e_v) {
      if (!grep /^$v-/ || /-$v\Z/, @E2) { push @E2, $v }
    }
    my $o2 = Graph->new(E => \@E2);
    my $gamma2 = $o2->gamma;
    $edge_removal_gamma_diff{$e} = $gamma2 - $gamma;
  }

  $o->{edge_removal_gamma_diff} = { %edge_removal_gamma_diff };
  return %edge_removal_gamma_diff;
}

sub is_UER {
  my $o = shift;
```

```
  my %edge_removal_gamma_diff = $o->edge_removal_gamma_diff;
  for (values %edge_removal_gamma_diff) {
    return 0 if $_ != 0;
  }
  return 1;
}

sub is_CER {
  my $o = shift;
  my %edge_removal_gamma_diff = $o->edge_removal_gamma_diff;
  for (values %edge_removal_gamma_diff) {
    return 0 if $_ == 0;
  }
  return 1;
}

sub E_all {
# returns a list of all possible edges for G.
  my $o = shift;
  my @V = $o->V;
  my @E_all;
  for my $i1 (0 .. $#V - 1) {
    for my $i2 ($i1 + 1 .. $#V) {
      push @E_all, "$V[$i1]-$V[$i2]";
    }
  }
  return @E_all;
}

sub edge_addition_gamma_diff {
# returns a hash with edge ab as key and
# gamma(G) - gamma(G + ab) as value.  the value
# is either 0 or 1.
  my $o = shift;
  return %{$o->{edge_addition_gamma_diff}}
      if %{$o->{edge_addition_gamma_diff}};
  my $gamma = $o->gamma;
  my %edge_addition_gamma_diff;
  my @E = $o->E;
  my @E_all = $o->E_all;
  my @E_complement = list_minus(\@E_all, \@E);
  for my $e (@E_complement) {
    my @E2 = (@E, $e);
    my $o2 = Graph->new(E => \@E2);
    my $gamma2 = $o2->gamma;
    $edge_addition_gamma_diff{$e} = $gamma - $gamma2;
  }
  $o->{edge_addition_gamma_diff} = { %edge_addition_gamma_diff };
  return %edge_addition_gamma_diff;
}

sub is_UEA {
  my $o = shift;
  my %edge_addition_gamma_diff = $o->edge_addition_gamma_diff;
  for (values %edge_addition_gamma_diff) {
```

```
    return 0 if $_ != 0;
  }

  return 1;
}

sub is_CEA {
  my $o = shift;
  my %edge_addition_gamma_diff = $o->edge_addition_gamma_diff;
  for (values %edge_addition_gamma_diff) {
    return 0 if $_ == 0;
  }

  return 1;
}

sub vertex_removal_gamma_diff {
# returns a hash with vertex v as key and
# gamma(G) - gamma(G - v) as value.  the value
# is -1, 0, or any positive integer.
  my $o = shift;
  return %{$o->{vertex_removal_gamma_diff}}
      if %{$o->{vertex_removal_gamma_diff}};

  my $gamma = $o->gamma;
  my %vertex_removal_gamma_diff;
  my @V = $o->V;
  my @E = $o->E;
  for my $v (@V) {
    my @E2 = grep !/^$v-/ && !/-$v\Z/, @E;  # remove edges containing $v.
    @E2 = grep $_ ne $v, @E2;  # remove $v as an isolate.
    @E2 = sort @E2;
    my $o2 = Graph->new(E => \@E2);
    my $gamma2 = $o2->gamma;
    $vertex_removal_gamma_diff{$v} = $gamma - $gamma2;
    my $diff = $gamma - $gamma2;
  }

  $o->{vertex_removal_gamma_diff} = { %vertex_removal_gamma_diff };
  return %vertex_removal_gamma_diff;
}

sub is_UVR {
  my $o = shift;
  my %vertex_removal_gamma_diff = $o->vertex_removal_gamma_diff;
  for (values %vertex_removal_gamma_diff) {
    return 0 if $_ != 0;
  }

  return 1;
}

sub is_CVR {
  my $o = shift;
  my %vertex_removal_gamma_diff = $o->vertex_removal_gamma_diff;
  for (values %vertex_removal_gamma_diff) {
    return 0 if $_ == 0;
  }

  return 1;
```

```
}

sub gamma_changing_unchanging_region {
  my $o = shift;
  return $o->{gamma_changing_unchanging_region}
      if $o->{gamma_changing_unchanging_region};

  my $UER = $o->is_UER;
  my $CER = $o->is_CER;
  my $UEA = $o->is_UEA;
  my $CEA = $o->is_CEA;
  my $UVR = $o->is_UVR;
  my $CVR = $o->is_CVR;

  my $gamma_changing_unchanging_region =
      ($UEA && !$UER && !$UVR && !$CER) ? 'R1'
    : ($UVR && !$UER && !$CER)          ? 'R2'
    : ($UER && $UEA && !$UVR)           ? 'R3'
    : ($UER && $UVR)                    ? 'R4'
    : ($UVR && $CER)                    ? 'R5'
    : ($UEA && $CER && !$UVR)           ? 'R6'
    : ($UER && !$UEA && !$CEA && !$CVR) ? 'R8'
    : ($CER && !$UEA && !$CEA)          ? 'R9'
    : ($UER && $CVR && !$CEA)           ? 'R10'
    : ($UER && $CVR && $CEA)            ? 'R11'
    : ($UER && $CEA && !$CVR)           ? 'R12'
    : ($CEA && !$UER && !$CER)          ? 'R13'
    : ($CER && $CEA)                    ? 'R14'
    :                                     'R7';

  $o->{gamma_changing_unchanging_region} = $gamma_changing_unchanging_region;
  return $gamma_changing_unchanging_region;
}

sub is_beta_0_set {
# returns 1 if the vertex set @v is independent; 1 otherwise.
  my $o = shift;
  my @v = @_;

  my $induced_subgraph = $o->induced_subgraph(@v);
  my @E = $induced_subgraph->E;
  return 0 if grep /-/, @E;

  return 1;
}

sub beta_0 {
# returns the vertex independence number.
  my $o = shift;
  my %param = @_;
  if (defined $o->{beta_0}) {
    return($o->{beta_0}, @{$o->{beta_0_set}});
  }

  # find the total domination number.
  my @V = $o->V;
  my $subset_generator = subset_generator(\@V, reverse => 1);

  my $beta_0 = 0;
  my @beta_0_sets;

  while (my @subset = $subset_generator->()) {
```

```
    if ($o->is_beta_0_set(@subset)) {
      last if $beta_0 and @subset < $beta_0;
      $beta_0 = @subset;
      push @beta_0_sets, [ @subset ];
      last if !exists $param{max};
      last if exists $param{max} and @beta_0_sets >= $param{max};
    }
  }

  $o->{beta_0} = $beta_0;
  $o->{beta_0_set} = $beta_0_sets[0];
  return wantarray ? ($o->{beta_0}, @beta_0_sets) : $o->{beta_0};
}

sub queen_graph_diag_vertices {
# returns the vertices within $diff distance of the main diagonal.
  my $o = shift;
  my $diff = shift || 0;
  my @V = $o->V;
  my @v;
  for my $v (@V) {
    my($row, $col) = $v =~ /^([a-z]+)(\d+)$/  or die "can't match v($v)!";
    my $row_i = ord($row) - ord('a') + 1;
    next if abs($row_i - $col) > $diff;
    push @v, $v;
  }
  return @v;
}

sub queen_graph_border_vertices {
# returns the vertices within $diff distance of the main diagonal.
  my $o = shift;
  my @V = $o->V;
  my $n = $o->n;
  my $col_max = sqrt($n);
  my $row_max = chr(ord('a') + sqrt($n) - 1);

  my @v;
  for my $v (@V) {
    my($row, $col) = $v =~ /^([a-z]+)(\d+)$/  or die "can't match v($v)!";
    if ($row eq 'a' or $row eq $row_max or
        $col == 1 or $col == $col_max) {
      push @v, $v;
    }
  }
  return @v;
}

sub queen_graph_corner_vertices {
# returns the vertices in the NW and SE corners of the queen's graph.
  my $o = shift;
  my $i = shift || 0;

  my @V = $o->V;
  my $n = $o->n;
  my $col_max = sqrt($n);
  my $row_max = chr(ord('a') + sqrt($n) - 1);

  my @v;
```

```
  for my $v (@V) {
    my($row, $col) = $v =~ /^([a-z]+)(\d+)$/  or die "can't match v($v)!";
    my $row_i = ord($row) - ord('a') + 1;
    if ($row_i + $col <= $i + 1 or
        $row_i + $col >= 2 * $col_max - $i + 1) {
      push @v, $v;
    }
  }
  # in case we have Qu1.
  if (!@v) { @v = ('a1') }
  return @v;
}
sub queen_graph_symmetry_set {
  my $o = shift;
  my @v = @_;
  my @v_sym = (
    [@v],
    [$o->queen_graph_rotation(@v)],
    [$o->queen_graph_rotation($o->queen_graph_rotation(@v))],
    [$o->queen_graph_rotation($o->queen_graph_rotation(
     $o->queen_graph_rotation(@v)))],
    [$o->queen_graph_reflection(@v)],
    [$o->queen_graph_reflection($o->queen_graph_rotation(@v))],
    [$o->queen_graph_reflection($o->queen_graph_rotation(
     $o->queen_graph_rotation(@v)))],
    [$o->queen_graph_reflection($o->queen_graph_rotation(
     $o->queen_graph_rotation($o->queen_graph_rotation(@v))))],
  );
  @v_sym = map [sort @$_], @v_sym;
  @v_sym = sort uniq(map "@$_", @v_sym);
  return @v_sym;
}
sub queen_graph_rotation {
  my $o = shift;
  my @v = @_;
  my @v2;
  my $n = $o->n;
  for my $v (@v) {
    my($row, $col) = $v =~ /^([A-Za-z]+)(\d+)$/ or die "can't match v($v)!";
    my $row2 = chr(ord("a") + $col - 1);
    my $col2 = sqrt($n) - (ord($row) - ord("a"));
    push @v2, "$row2$col2";
  }
  return @v2;
}
sub queen_graph_reflection {
  my $o = shift;
  my @v = @_;
  my @v2;
  my $n = $o->n;
  for my $v (@v) {
    my($row, $col) = $v =~ /^([A-Za-z]+)(\d+)$/ or die "can't match v($v)!";
```

```
    my $row2 = chr(ord("a") + $col - 1);
    my $col2 = 1 + (ord($row) - ord("a"));
    push @v2, "$row2$col2";
  }
  return @v2;
}


1;
```

Graph/Tree.pm: a module implementing a tree as an object.

```
# created: sml/2005-02-03
# updated: sml/2006-01-14
# Graph/Tree.pm - a tree is a connected graph with no cycles.
use strict;
use warnings;
use lib '/home/sml/lib';

package Graph::Tree;
my $CLASS = 'Graph::Tree';

use Graph;
our @ISA = qw(Graph);

use Graph::TreePicture;  # the picture-making subs

use SML qw(list_minus uniq);

### CLASS METHODS

sub new {
  my $class = shift;
  my $o = $class->SUPER::new(@_);

  $o->{leaves} = [];   # the vertices that are leaves of the tree.
  $o->{support} = {};  # the vertices that are support vertices of the tree.
                       # the value is the number of leaves the vertex supports;
                       # a vertex is a strong support vertex if the value > 1.

  # rooted-tree stuff.
  $o->{root} = undef;
  $o->{v_level} = {};  # the level that the vertex is on;
                       # the distance from the root.
  $o->{levels} = [];   # vertices on each level.
  $o->{v_parent} = {}; # the parent of each vertex.
  $o->{v_kids} = {};  # the kids of each vertex.

  $o->{canon} = undef;  # the canonical string representing the tree.

  bless $o, $class;
  $o->init;
  return $o;
}

### OBJECT METHODS

sub init {
  my $o = shift;
  $o->SUPER::init;

  # trees always have these params.
  $o->{ecc} = {};
  $o->{diam} = undef;
  $o->{rad} = undef;
  $o->{center} = [];
  $o->{periphery} = [];

  # find the eccentricity params.
  my @center = $o->center;

  return $o;
}

sub center {
# find the center of the tree.
```

```
# (and the radius and diameter.)
  my $o = shift;
  my @center = @{$o->{center}};
  return @center if @center;

  my @V = $o->V;
  my @v = @V;
  my @e = $o->E;

  my $rad = 0;
  my @v_last;
  {
    #warn "V(@v) E(@e)";
    @v == @v_last and die "hmm? (@v) = (@v_last)";
    @v_last = @v;
    last if @v == 1 or @v == 2;  # trees' centers always have 1 or 2 vertices.
    my @leaves;
    for my $v (@v) {
      my $deg = grep /$v/, @e;
      $deg == 1 and push @leaves, $v;
    }
    my @e_new;
    for my $e (@e) {
      push @e_new, $e if !grep $e =~ /^$_-/ || $e =~ /-$_$/, @leaves;
    }
    @e = @e_new;
    $rad++;
    @v = list_minus(\@v, \@leaves);
    redo;
  }

  @center = sort @v;

  $rad += @v - 1;
  $o->{rad} = $rad;
  $o->{center} = [ @center ];

  my $diam = 2 * $rad - int(@v / 2);
  $o->{diam} = $diam;

  return @center;
}

sub ecc {
  my $o = shift;
  my %ecc = %{$o->{ecc}};

  if (!%ecc) {
    # then calculate it.
    my @center = $o->center;
    my @v = $o->V;
    my $rad = $o->rad;
    my $ecc = $rad;

    my @No = @center;
    my @v_used;
    {
      @ecc{@No} = ($ecc)x@No;
      @v = list_minus(\@v, \@No);
      last if !@v;

      $ecc++;
```

```
      push @v_used, @No;
      @No = map $o->No($_), @No;
      @No = list_minus(\@No, \@v_used);
      redo;
    }
    my $diam = $o->diam;
    $diam == $ecc or die "diam doesn't match! ($diam vs $ecc)";
  }

  return %ecc if wantarray;
  my $ecc = join ' ', map "$_:$ecc{$_}", $o->V;
  return $ecc;
}

sub leaves {
  my $o = shift;
  my @leaves = @{$o->{leaves}};
  return @leaves if @leaves;

  my %deg = $o->deg;
  my @V = $o->V;
  @leaves = grep $deg{$_} == 1, @V;
  $o->{leaves} = [ @leaves ];
  return @leaves;
}

sub support {
  my $o = shift;
  my %support = %{$o->{support}};
  return %support if %support;

  my @leaves = $o->leaves;
  my @support = map $o->No($_), @leaves;
  for my $support (@support) {
    $support{$support}++;
  }

  $o->{support} = { %support };
  return %support;
}

sub periphery {
  my $o = shift;
  my @periphery = @{$o->{periphery}};
  return @periphery if @periphery;
  my %ecc = $o->ecc;
  my $diam = $o->diam;
  @periphery = sort grep $ecc{$_} == $diam, keys %ecc;
  $o->{periphery} = [ @periphery ];
  return @periphery;
}

sub geodesic {
# returns vertices V that make a geodesic of the tree;
# <V> = a path of length diam(T).
# here's the algorithm:
# - pick a vertex from the periphery.
# - keep choosing neighbors until a center vertex is reached.
# - this makes the first arm of the geodesic.
```

```
# - pick another vertex from the periphery.
# - keep choosing neighbors until a center vertex is reached,
#   with the requirements that:
#   - if a vertex in the first arm is reached, start with a
#     different periphery, unless the vertex reached is in the
#     center, and the center has only one vertex,
# if we do it right, we end up with diam(T) + 1 vertices.
  my $o = shift;
  my @geodesic = @{ $o->{geodesic} || []};
  return @geodesic if @geodesic;
  my %ecc = $o->ecc;
  my $diam = $o->diam;
  my @center = $o->center;
  my @periphery = $o->periphery;

  # first arm.
  my $v = shift @periphery;
  {
    push @geodesic, $v;
    last if grep $v eq $_, @center;
    ($v) = grep $ecc{$_} == $ecc{$v} - 1, $o->No($v);
    redo;
  }

  # special case for K1.
  $diam == 0 and return @geodesic;

  # second arm.
  # two cases; @center == 1 or @center == 2.
  if (@center == 1) {
    my $center = shift @center;
    ARM: {
      my @arm;
      !@periphery and die "oops, ran out of periphery!";
      $v = shift @periphery;
      V: {
        push @arm, $v;
        ($v) = grep $ecc{$_} == $ecc{$v} - 1, $o->No($v);
        !defined($v) and die "oops, can't find a v!";
if ($v eq $center) {
  push @geodesic, reverse @arm;
  last ARM;
}
        redo ARM if grep $v eq $_, @geodesic;
        redo V;
      }
    }
  }

  # @center == 2.
  else {
    @center = grep $v ne $_, @center;
    my $center = shift @center;
    ARM: {
      my @arm;
      !@periphery and die "oops, ran out of periphery!";
      $v = shift @periphery;
      V: {
        push @arm, $v;
        if ($v eq $center) {
```

```
          push @geodesic, reverse @arm;
  last ARM;
        }
        ($v) = grep $ecc{$_} == $ecc{$v} - 1, $o->No($v);
        !defined($v) and die "oops, can't find a v!";
        redo ARM if grep $v eq $_, @geodesic;
        redo V;
      }
    }
  }

  @geodesic == $diam + 1 or die "oops, geodesic doen't equal $diam + 1!";

  $o->{geodesic} = [ @geodesic ];

  return @geodesic;
}

sub canon {
# we root the tree at its center, which is either 1 or 2 vertices.
  my $o = shift;
  my $canon = $o->{canon};
  return $canon if $canon;

  my @center = $o->center;

  if (@center == 2) {
    $o->root_at($center[0]);
    my $canon0 = $o->canonize;
    $o->root_at($center[1]);
    my $canon1 = $o->canonize;
    $canon = (sort by_canon $canon0, $canon1)[0];
  }
  else {
    $o->root_at($center[0]);
    $canon = $o->canonize;
  }

  $o->{canon} = $canon;
  return $canon;
}

sub canonize {
# given a rooted tree, return its canon.
  my $o = shift;

  # start from the bottom and work up.
  my %canon;
  my $level_max = @{$o->{levels}} - 1;
  for my $level (reverse 0 .. $level_max) {
    my @v = @{$o->{levels}[$level]};
    #warn "LEVEL $level: @v";
    for my $v (@v) {
      my @kids = @{$o->{v_kids}{$v}};
      if (!@kids) {
        $canon{$v} = 1;
      }
      else {
        if (@kids == 1 and $canon{$kids[0]} =~ /^\d+$/) {
  $canon{$v} = $canon{$kids[0]} + 1;
}
```

```
else {
  $canon{$v} = '(' . join(',', sort by_canon @canon{@kids}) . ')';
}
      }
      #warn "v($v) canon[ $canon{$v} ]";
    }
  }

  my $canon = $canon{$o->{root}};
  if ($canon =~ /^\((.*)\)$/) {
    $canon = $1;
  }

  return $canon;
}

sub by_canon {
# may make this more complex later.
  $b cmp $a;
}

sub root_at {
# root the tree at vertex $v, and find its structure.
  my $o = shift;
  my $root = shift;
  my @v = $o->V;
  my @E = $o->E;

  $o->{root} = $root;
  my $level = 0;
  $o->{v_level}{$root} = $level;
  $o->{levels}[$level] = [ $root ];
  $o->{v_parent}{$root} = '';

  {
    $level++;
    my @level;
    for my $v (@{$o->{levels}[$level - 1]}) {
      my @kids = $o->No($v);
      @kids = grep $o->{v_parent}{$v} ne $_, @kids;
      $o->{v_kids}{$v} = [ @kids ];
      $o->{v_level}{$root} = $level;
      map $o->{v_parent}{$_} = $v, @kids;
      push @level, @kids;
    }
    last if !@level;
    $o->{levels}[$level] = [ @level ];
    redo;
  }

  return;
}

### SUBS

1;
```

Graph/TreePicture.pm: a module for making figures of trees.

```
# created: sml/2005-02-03
# updated: sml/2005-06-12
# Graph/TreePicture.pm - the picture-making subs for Graph::Tree.
use strict;
use warnings;
use lib '/home/sml/lib';

package Graph::Tree;
my $CLASS = 'Graph::Tree';

use SML qw(list_minus uniq);

### SUBS

sub pic_matrix {
  my $o = shift;
  my $rad = $o->rad;
  my $diam = $o->diam;
  my @geodesic = $o->geodesic;

  # the pic matrix has $diam + 1 == @geodesic cols,
  # and 2 * $rad + 1 rows.  the geodesic is initially
  # placed as the center row.
  my @pic;
  my %pic_vertices;
  for my $row (0 .. $rad - 1) {
    $pic[$row] = [ (undef)x@geodesic ];
  }
  for my $col (0 .. $#geodesic) {
    $pic[$rad][$col] = $geodesic[$col];
    $pic_vertices{$geodesic[$col]} = [ $rad, $col ];
  }
  for my $row ($rad + 1 .. 2 * $rad) {
    $pic[$row] = [ (undef)x@geodesic ];
  }

  # the order that we want to place neighbors:
  # N, S, E, W, NE, SE, NW, SW.
  my @dir = (
    [-1, 0], [1, 0], [0, 1], [0, -1],
    [-1, 1], [1, 1], [-1, -1], [1, -1]
  );

  # place the vertices.
  my @V = $o->V;
  my @v_placed = @geodesic;
  my @v_last = @geodesic;
  my %dir_history;
  while (@v_placed != @V) {
    my @v_placed_new;
    for my $v (@v_last) {
      my($row, $col) = @{$pic_vertices{$v}};
      # find the unplaced neighbors of $v.
      my @No = $o->No($v);
      @No = list_minus(\@No, \@v_placed);
      for my $w (@No) {
        for my $dir ($dir_history{$v}, @dir) {
```

```
  next if !defined $dir;
  my $row_new = $row + $dir->[0];
  my $col_new = $col + $dir->[1];
  next if defined $pic[$row_new][$col_new];
  $pic[$row_new][$col_new] = $w;
  $pic_vertices{$w} = [ $row_new, $col_new ];
  $dir_history{$w} = $dir;
  last;
}
if (!$pic_vertices{$w}) {
  # XXX handle more gracefully.
  die "oops, couldn't place vertex($w)!";
}
      }
      push @v_placed_new, @No;
    }
    push @v_placed, @v_placed_new;
    @v_last = @v_placed_new;
  }

  # clear out empty rows at the top and bottom.
  my $rows = @pic;
  my $empty = 0;
  for my $row (0 .. $rows - 1) {
    last if grep defined($_), @{$pic[$row]};
    $empty++;
  }
  @pic = @pic[$empty .. $rows - 1];
  for my $v (keys %pic_vertices) {
    my($row, $col) = @{$pic_vertices{$v}};
    $row -= $empty;
    $pic_vertices{$v} = [ $row, $col ];
  }
  $empty = 0;
  $rows = @pic;
  for my $row (reverse 0 .. $rows - 1) {
    last if grep defined($_), @{$pic[$row]};
    $empty++;
  }
  @pic = @pic[0 .. $rows - 1 - $empty];

  return \@pic, \%pic_vertices;
}

sub ascii_pic {
  my $o = shift;
  my %param = @_;
  my @S = @{$param{S} || []};

  # get the pic matrix.
  my($pic_matrix, $pic_vertices) = $o->pic_matrix;
  my %pic_vertices = %$pic_vertices;

  # double the matrix, to add edges.
  my @pic_matrix_edges;
  my %pic_vertices_edges;
  for my $v (keys %{$pic_vertices}) {
    my($row, $col) = @{$pic_vertices{$v}};
```

```
    my $label = grep($_ eq $v, @S) ? uc($v) : $v;
    $pic_matrix_edges[$row * 2][$col * 2] = $label;
    $pic_vertices_edges{$v} = [ $row * 2, $col * 2 ];
  }

  # add the edges.
  my @E = $o->E;
  my %edge_label;
  $edge_label{-2}{0} = $edge_label{2}{0} = '|';
  $edge_label{0}{-2} = $edge_label{0}{2} = '-';
  $edge_label{-2}{2} = $edge_label{2}{-2} = '/';
  $edge_label{2}{2} = $edge_label{-2}{-2} = '\\';
  for my $e (@E) {
    next unless my($v1, $v2) = $e =~ /^([^-]+)-([^-]+)$/;
    my($row1, $col1) = @{$pic_vertices_edges{$v1}};
    my($row2, $col2) = @{$pic_vertices_edges{$v2}};

    my $rowdiff = $row1 - $row2;
    my $coldiff = $col1 - $col2;
    my $entry = $edge_label{$rowdiff}{$coldiff}
      or die "no edge label for rowdiff($rowdiff) coldiff($coldiff)!";

    my $rowavg = ($row1 + $row2) / 2;
    my $colavg = ($col1 + $col2) / 2;

    $pic_matrix_edges[$rowavg][$colavg] = $entry;
  }

  # get the dimensions.
  my $rows = @pic_matrix_edges;
  my @geodesic = $o->geodesic;
  my $cols = @geodesic * 2 + 1;

  # render the ASCII pic.
  my $pic = '';
  for my $row (0 .. $rows - 1) {
    for my $col (0 .. $cols - 1) {
      my $entry = $pic_matrix_edges[$row][$col];
      $pic .= defined($entry) ? $entry : ' ';
    }
    $pic .= "\n";
  }

  return $pic;
}

sub latex_pic {
  my $o = shift;
  my %param = @_;
  my @S = @{$param{S} || []};
  my $n = $o->n;
  my $diam = $o->diam;
  my $caption = $param{caption} || "\$n = $n, diam = $diam\$\n";

  # get the pic matrix.
  my($pic_matrix, $pic_vertices) = $o->pic_matrix;
  my %pic_vertices = %$pic_vertices;

  # get the dimensions.
```

```
  my $rows = @$pic_matrix;
  my @geodesic = $o->geodesic;
  my $cols = @geodesic;

  # separation between vertices.
  my $sep = $param{sep} || 8;

  # start.
  my $latex = '';
  #$latex .= "\\begin{figure}\n";
  #$latex .= "\\begin{center}\n";
  #$latex .= "\\itshape\n";

  my $width = $cols * $sep + 10;
  my $height = $rows * $sep + 10;
  $latex .= "\\begin{picture}($width,$height)(0,0)\n";

  # add the vertices.
  for my $v (keys %pic_vertices) {
    my($row, $col) = @{$pic_vertices{$v}};
    my $command = grep($v eq $_, @S) ? 'vertex' : 'vertexo';
    my $x = $col * $sep + 5;
    my $y = ($rows - $row) * $sep + 5;
    $latex .= "\\$command($x,$y)\n";
  }

  # add the edges.
  my @E = $o->E;
  for my $e (@E) {
    next unless my($v1, $v2) = $e =~ /^([^-]+)-([^-]+)$/;
    my($row, $col) = @{$pic_vertices{$v1}};
    my($row2, $col2) = @{$pic_vertices{$v2}};

    my $rowdiff = $row2 - $row;
    $rowdiff *= -1;  # flipping coords to match latex.
    my $coldiff = $col2 - $col;

    my $x = $col * $sep + 5;
    my $y = ($rows - $row) * $sep + 5;
    $latex .= "\\put($x,$y){\\line($coldiff,$rowdiff){$sep}}\n";
  }

  # details.
  $latex .= $caption;

  # finish.
  $latex .= "\\end{picture}\n";
  #$latex .= "\\end{center}\n";
  #$latex .= "\\caption{All ULD-trees of Order $\\le 15$}\n";
  #$latex .= "\\label{f:all_uld_trees_15}\n";
  #$latex .= "\\end{figure}\n";

  return $latex;
}

1;
```

tree-domination-numbers: a "driver" using the above modules to find (among other things) ULD-trees.

```
#!/usr/bin/perl -w
# tree-domination-numbers
# find domination numbers (and other stuff) about trees.
use strict;

use lib '/home/sml/lib';
use Graph;

my %GRAPH_DEFS = (
  '1.1'  => [ qw(a) ],
  '2.1'  => [ qw(ab) ],
  '3.1'  => [ qw(ab bc) ],
  '4.1'  => [ qw(ab bc cd) ],
  '4.2'  => [ qw(ab bc bd) ],
  '5.1'  => [ qw(ab bc cd de) ],
  '5.2'  => [ qw(ab bc cd ce) ],
  '5.3'  => [ qw(ab bc bd be) ],
  '6.1'  => [ qw(ab bc cd de ef) ],
  '6.2'  => [ qw(ab bc cd de df) ],
  '6.3'  => [ qw(ab bc cd be ef) ],
  '6.4'  => [ qw(ab bc bd de df) ],
  '6.5'  => [ qw(ab bc cd ce cf) ],
  '6.6'  => [ qw(ab bc bd be bf) ],
  '7.1'  => [ qw(ab bc cd de ef fg) ],
  '7.2'  => [ qw(ab bc cd de ef eg) ],
  '7.3'  => [ qw(ab bc cd de df fg) ],
  '7.4'  => [ qw(ab cb bd de ef eg) ],
  '7.5'  => [ qw(ab bc cd de df dg) ],
  '7.6'  => [ qw(ab cb bd de df fg) ],
  '7.7'  => [ qw(ab bc cd ce cf fg) ],
  '7.8'  => [ qw(ab bc cd de cf fg) ],
  '7.9'  => [ qw(ab cb bd de df dg) ],
  '7.10' => [ qw(ab bc cd ce cf cg) ],
  '7.11' => [ qw(ab ac ad ae af ag) ],
  '8.1'  => [ qw(ab bc cd de ef fg gh) ],
  '8.2'  => [ qw(ab bc cd de ef fg fh) ],
  '8.3'  => [ qw(ab bc cd de ef eg gh) ],
  '8.4'  => [ qw(ab cb bd de ef fg fh) ],
  '8.5'  => [ qw(ab bc cd de bf fg gh) ],
  '8.6'  => [ qw(ab bc cd de ef eg eh) ],
  '8.7'  => [ qw(ab cb bd de ef eg gh) ],
  '8.8'  => [ qw(ab bc bd de df fg gh) ],
  '8.9'  => [ qw(ab bc bd de ef eg eh) ],
  '8.10' => [ qw(ab bc cd de ef dg gh) ],
  '8.11' => [ qw(ab bc cd ce ef eg gh) ],
  '8.12' => [ qw(ab bc cd de df dg gh) ],
  '8.13' => [ qw(ab bc bd de df fg fh) ],
  '8.14' => [ qw(ab bc cd de df dg dh) ],
  '8.15' => [ qw(ab bc cd ce ef eg eh) ],
  '8.16' => [ qw(ab bc bd de ef dg gh) ],
  '8.17' => [ qw(ab bc bd de df dg gh) ],
  '8.18' => [ qw(ab bc bd be ef eg eh) ],
```

```
  '8.19' => [ qw(ab bc bd de df dg dh) ],
  '8.20' => [ qw(ab bc cd ce ef cg gh) ],
  '8.21' => [ qw(ab bc bd be ef bg gh) ],
  '8.22' => [ qw(ab bc cd ce cf cg ch) ],
  '8.23' => [ qw(ab ac ad ae af ag ah) ],
 '17.1'  => [ qw(ab bc cd de ef fg gh hi ij jk kl lm mn no op pq) ],
 '18.1'  => [ qw(ab bc cd de ef fg gh hi ij jk kl lm mn no op pq qr) ],
 '19.1'  => [ qw(ab bc cd de ef fg gh hi ij jk kl lm mn no op pq qr rs) ],
 '20.1'  => [ qw(ab bc cd de ef fg gh hi ij jk kl lm mn no op pq qr rs st) ],
);

# note: the header below is truncated.
my $HEADER = <<EOF;
--------+----+----+----+---------------
ID      |  g | gL | gD | gamma_sets
--------+----+----+----+---------------
EOF

print $HEADER;

my $count = 0;
for my $name (sort by2levels keys %GRAPH_DEFS) {
  my($n, $id) = split /\./, $name;
  #next unless $n >= 16 and $id >= 262;
  $count++;
  print $HEADER if $count % 30 == 0;
  my @E = @{ $GRAPH_DEFS{$name} };
  my @E_vertices = map join('-', split //, $_), @E;
  my $graph = Graph->new(
    name => $name,
    E => \@E_vertices,
  );
  #print $graph->info;
  my($gamma, @gamma_sets) = $graph->gamma(max => 2);
  my $gamma_set_count = @gamma_sets;
  my $gamma_sets;
  {
    local $" = '';
    $gamma_sets = $gamma_set_count <= 2
      ? join(' ', map "(@$_)", @gamma_sets)
      : "(@{$gamma_sets[0]}) ...";
  }

  my($gamma_L, @gamma_L_sets) = $graph->gamma_L(max => 2);
  my $gamma_L_set_count = @gamma_L_sets;
  my $gamma_L_sets;
  {
    local $" = '';
    $gamma_L_sets = $gamma_L_set_count <= 2
      ? join(' ', map "(@$_)", @gamma_L_sets)
      : "(@{$gamma_L_sets[0]}) ...";
  }

  my($gamma_D, @gamma_D_sets) = $graph->gamma_D(max => 2);
  my $gamma_D_set_count = @gamma_D_sets;
  my $gamma_D_sets;
  {
    local $" = '';
```

```
    $gamma_D_sets = $gamma_D_set_count <= 2
      ? join(' ', map "(@$_)", @gamma_D_sets)
      : "(@{$gamma_D_sets[0]}) ...";
  }

  printf "%-7s | %2d | %2d | %2d | %3dg: %-25s | "
       . "%3dLg: %-25s | %3dDg: %-25s | E(@E)\n",
    $name, $gamma, $gamma_L, $gamma_D,
      $gamma_set_count, $gamma_sets,
      $gamma_L_set_count, $gamma_L_sets,
      $gamma_D_set_count, $gamma_D_sets;
}

exit;

sub by2levels {
  my @a = split /\./, $a;
  my @b = split /\./, $b;
  $a[0] <=> $b[0]
    or
  $a[1] <=> $b[1];
}
```

VITA

Stephen M. Lane

430 Lamont St.

Johnson City, TN 37604

sml@usually.com

Education

- Bachelors of Science Degree in Chemical Engineering and Biochemistry, University of Tennessee, May 1992.
- Masters of Science Degree in Mathematics, East Tennessee State University, May 2006.

Professional Experience

- Graduate Assistant / Teaching Assistant, Math Lab Student Director, East Tennessee State University, Johnson City, TN, 2004-2006.